CS6700 - RL

Dueling DQN results

Dueling DQN Introduction =>

Main characteristics of dueling DQN architecture is that it separates the state value representation and state-dependent action advantages via two separate streams. The main motivation behind this type of architecture is there are some environments in which it is unnecessary to know the value of each action at every timestep. By separating the two streams, dueling architecture can learn which states (or are not) valuable, without having to learn the effect of each action for each state.

In the dueling DQN architecture paper there are two different types of update equation :

1. **Type 1 update (mean of advantage function) =>**

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a;\theta,\alpha))$$

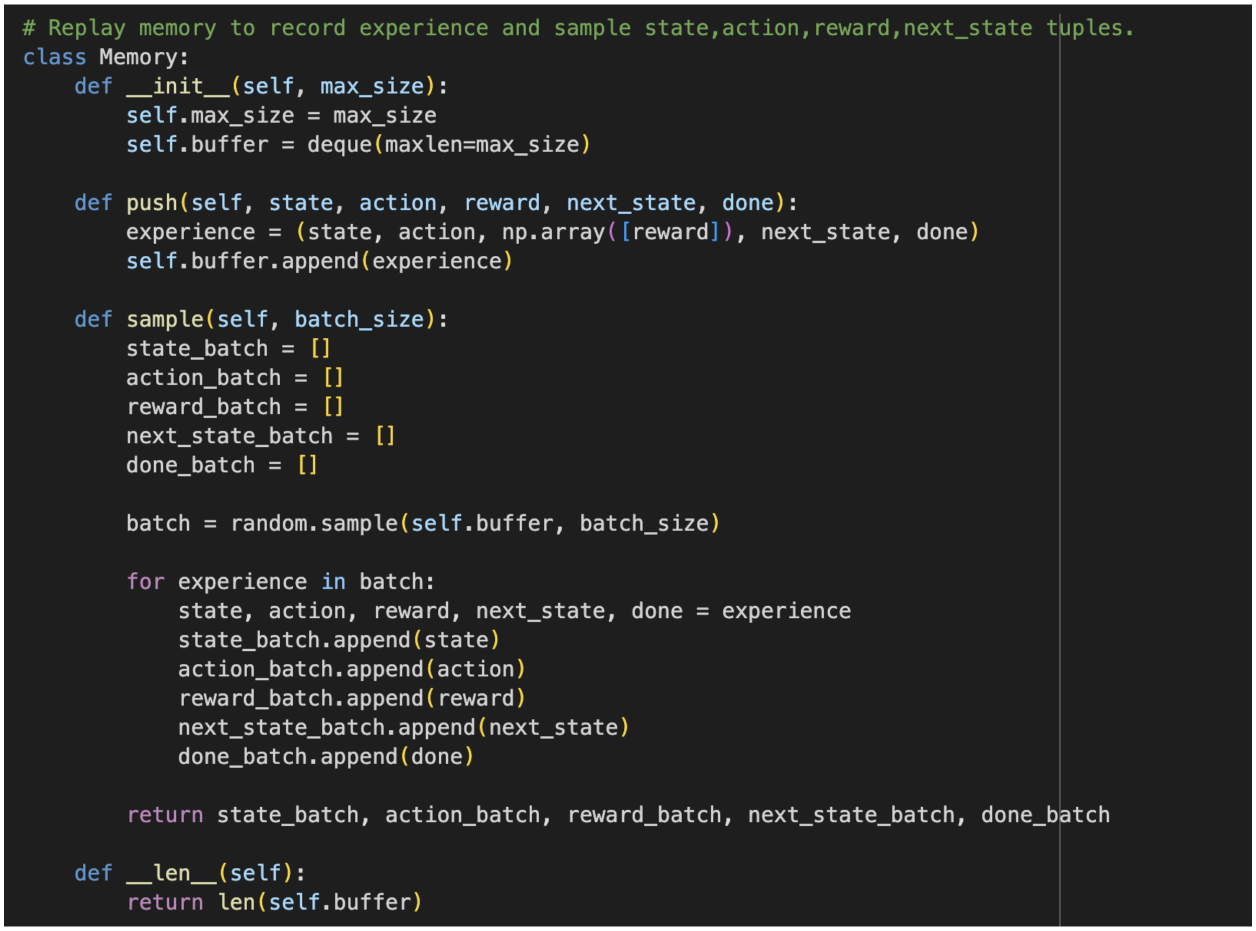
2. **Type 2 update (max of advantage function) =>**

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \max_{a' \in |\mathcal{A}|} A(s,a;\theta,\alpha))$$

Here we are trying to compare two different types of updates in Acrobat and Cartpole environments.

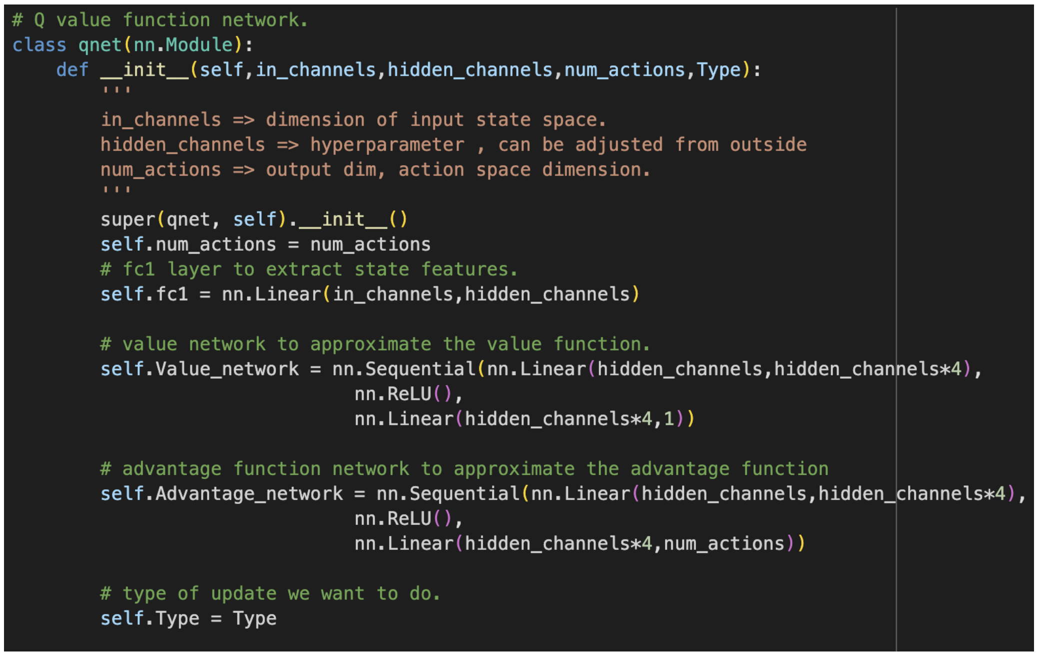
Code Snippets :

1. **Memory to store and sample required experiences =>**

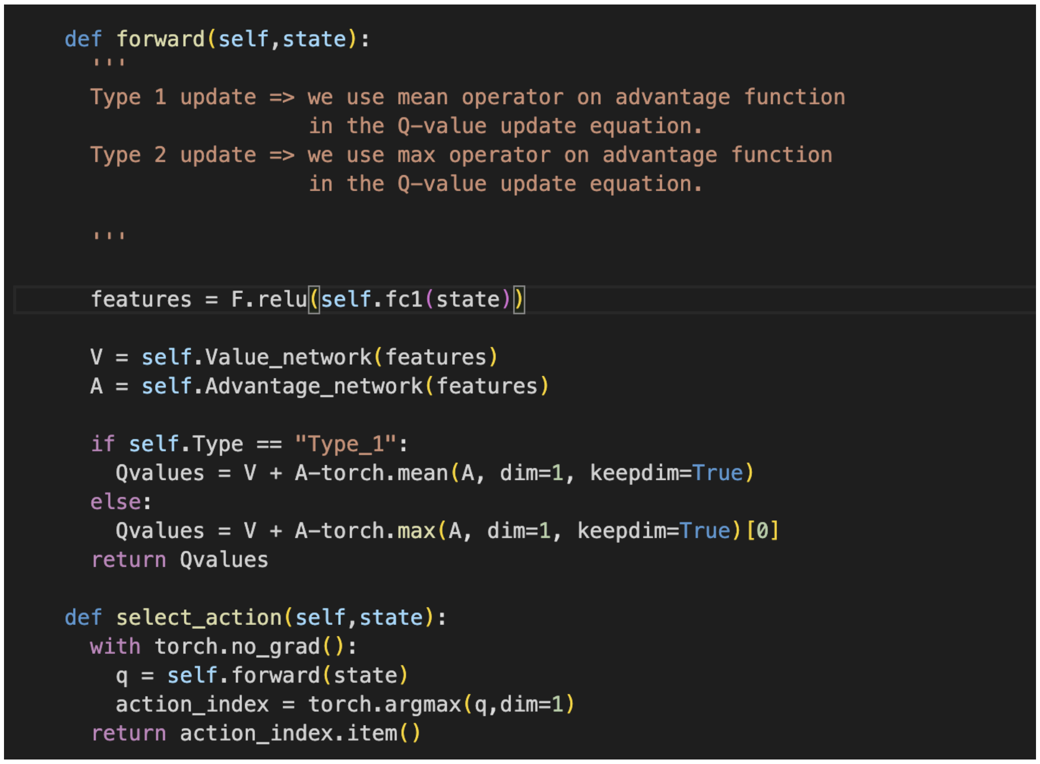


```python
# Replay memory to record experience and sample state,action,reward,next_state tuples.
class Memory:
    def __init__(self, max_size):
        self.max_size = max_size
        self.buffer = deque(maxlen=max_size)

    def push(self, state, action, reward, next_state, done):
        experience = (state, action, np.array([reward]), next_state, done)
        self.buffer.append(experience)

    def sample(self, batch_size):
        state_batch = []
        action_batch = []
        reward_batch = []
        next_state_batch = []
        done_batch = []

        batch = random.sample(self.buffer, batch_size)

        for experience in batch:
            state, action, reward, next_state, done = experience
            state_batch.append(state)
            action_batch.append(action)
            reward_batch.append(reward)
            next_state_batch.append(next_state)
            done_batch.append(done)

        return state_batch, action_batch, reward_batch, next_state_batch, done_batch

    def __len__(self):
        return len(self.buffer)
```

2. Main Q-network for Q-value estimation =>

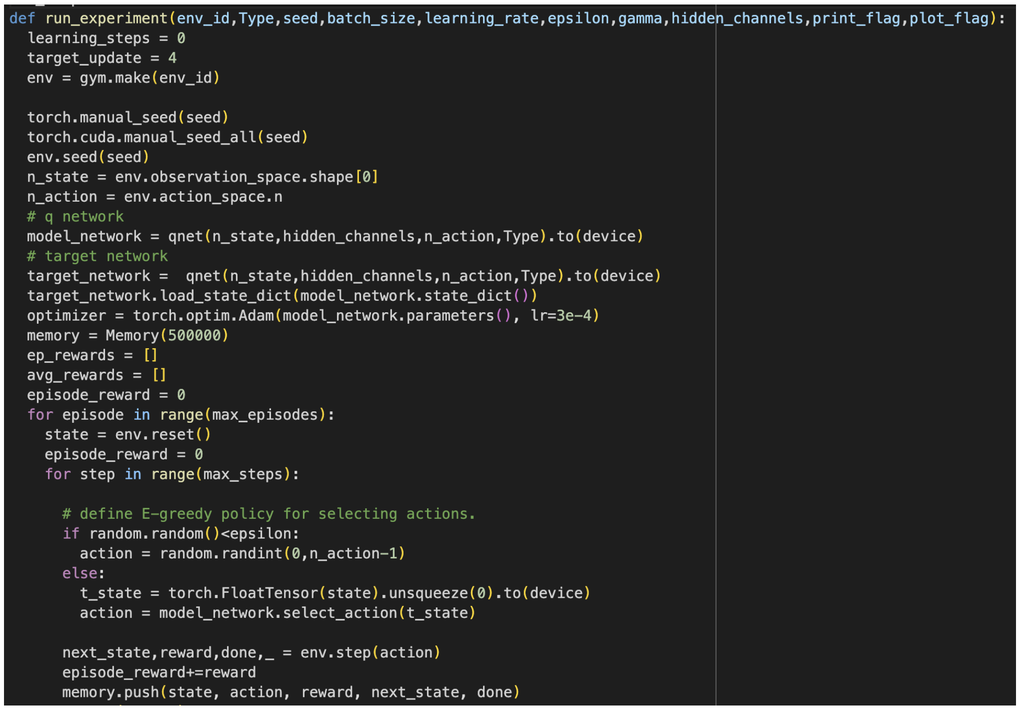


```python
# Q value function network.
class qnet(nn.Module):
    def __init__(self,in_channels,hidden_channels,num_actions,Type):
        '''
        in_channels => dimension of input state space.
        hidden_channels => hyperparameter , can be adjusted from outside
        num_actions => output dim, action space dimension.
        '''
        super(qnet, self).__init__()
        self.num_actions = num_actions
        # fc1 layer to extract state features.
        self.fc1 = nn.Linear(in_channels,hidden_channels)

        # value network to approximate the value function.
        self.Value_network = nn.Sequential(nn.Linear(hidden_channels,hidden_channels*4),
                               nn.ReLU(),
                               nn.Linear(hidden_channels*4,1))

        # advantage function network to approximate the advantage function
        self.Advantage_network = nn.Sequential(nn.Linear(hidden_channels,hidden_channels*4),
                               nn.ReLU(),
                               nn.Linear(hidden_channels*4,num_actions))

        # type of update we want to do.
        self.Type = Type
```

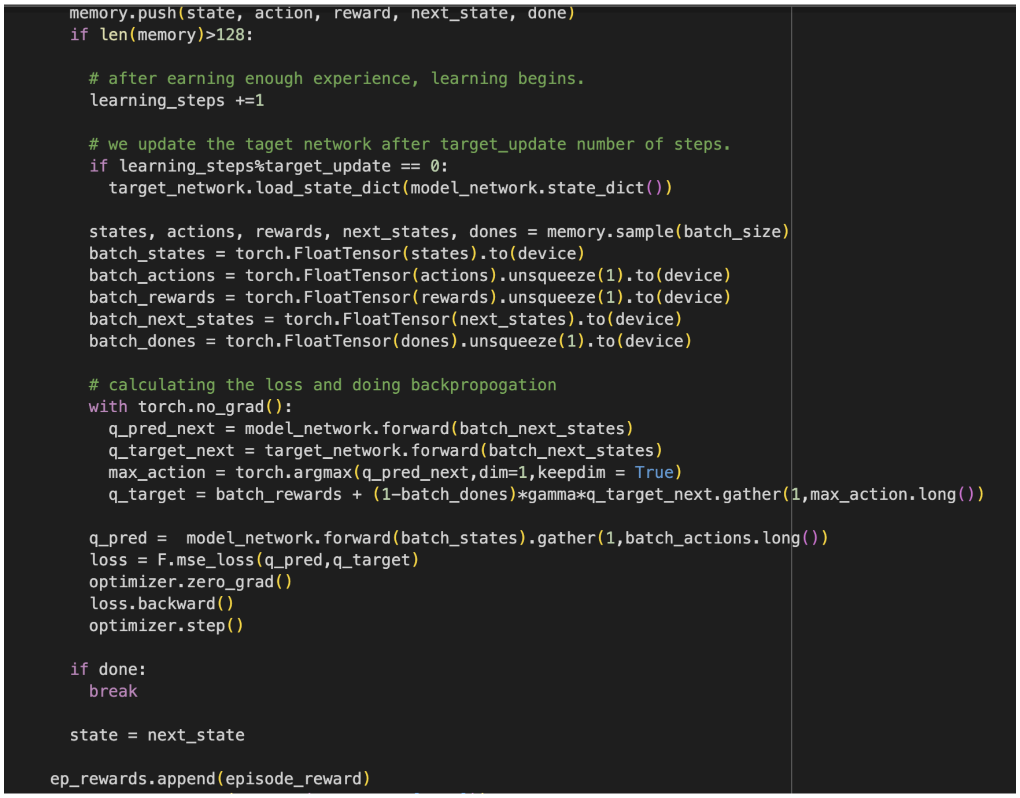


```python
    def forward(self,state):
      '''
      Type 1 update => we use mean operator on advantage function
                       in the Q-value update equation.
      Type 2 update => we use max operator on advantage function
                       in the Q-value update equation.

      '''

      features = F.relu(self.fc1(state))

      V = self.Value_network(features)
      A = self.Advantage_network(features)

      if self.Type == "Type_1":
        Qvalues = V + A-torch.mean(A, dim=1, keepdim=True)
      else:
        Qvalues = V + A-torch.max(A, dim=1, keepdim=True)[0]
      return Qvalues

    def select_action(self,state):
      with torch.no_grad():
        q = self.forward(state)
        action_index = torch.argmax(q,dim=1)
      return action_index.item()
```

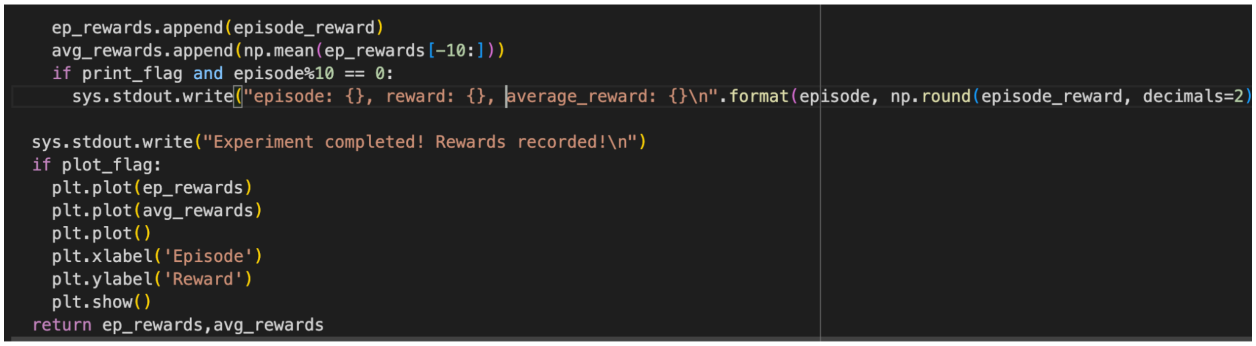
3. Main Learning Algorithm =>



```
def run_experiment(env_id,Type,seed,batch_size,learning_rate,epsilon,gamma,hidden_channels,print_flag,plot_flag):
  learning_steps = 0
  target_update = 4
  env = gym.make(env_id)

  torch.manual_seed(seed)
  torch.cuda.manual_seed_all(seed)
  env.seed(seed)
  n_state = env.observation_space.shape[0]
  n_action = env.action_space.n
  # q network
  model_network = qnet(n_state,hidden_channels,n_action,Type).to(device)
  # target network
  target_network =  qnet(n_state,hidden_channels,n_action,Type).to(device)
  target_network.load_state_dict(model_network.state_dict())
  optimizer = torch.optim.Adam(model_network.parameters(), lr=3e-4)
  memory = Memory(500000)
  ep_rewards = []
  avg_rewards = []
  episode_reward = 0
  for episode in range(max_episodes):
    state = env.reset()
    episode_reward = 0
    for step in range(max_steps):

      # define E-greedy policy for selecting actions.
      if random.random()<epsilon:
        action = random.randint(0,n_action-1)
      else:
        t_state = torch.FloatTensor(state).unsqueeze(0).to(device)
        action = model_network.select_action(t_state)

      next_state,reward,done,_ = env.step(action)
      episode_reward+=reward
      memory.push(state, action, reward, next_state, done)
```



```
      memory.push(state, action, reward, next_state, done)
      if len(memory)>128:

        # after earning enough experience, learning begins.
        learning_steps +=1

        # we update the taget network after target_update number of steps.
        if learning_steps%target_update == 0:
          target_network.load_state_dict(model_network.state_dict())

        states, actions, rewards, next_states, dones = memory.sample(batch_size)
        batch_states = torch.FloatTensor(states).to(device)
        batch_actions = torch.FloatTensor(actions).unsqueeze(1).to(device)
        batch_rewards = torch.FloatTensor(rewards).unsqueeze(1).to(device)
        batch_next_states = torch.FloatTensor(next_states).to(device)
        batch_dones = torch.FloatTensor(dones).unsqueeze(1).to(device)

        # calculating the loss and doing backpropogation
        with torch.no_grad():
          q_pred_next = model_network.forward(batch_next_states)
          q_target_next = target_network.forward(batch_next_states)
          max_action = torch.argmax(q_pred_next,dim=1,keepdim = True)
          q_target = batch_rewards + (1-batch_dones)*gamma*q_target_next.gather(1,max_action.long())

        q_pred =  model_network.forward(batch_states).gather(1,batch_actions.long())
        loss = F.mse_loss(q_pred,q_target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

      if done:
        break

      state = next_state

    ep_rewards.append(episode_reward)
```



```
    ep_rewards.append(episode_reward)
    avg_rewards.append(np.mean(ep_rewards[-10:]))
    if print_flag and episode%10 == 0:
      sys.stdout.write("episode: {}, reward: {}, average_reward: {}\n".format(episode, np.round(episode_reward, decimals=2)

  sys.stdout.write("Experiment completed! Rewards recorded!\n")
  if plot_flag:
    plt.plot(ep_rewards)
    plt.plot(avg_rewards)
    plt.plot()
    plt.xlabel('Episode')
    plt.ylabel('Reward')
    plt.show()
  return ep_rewards,avg_rewards
```

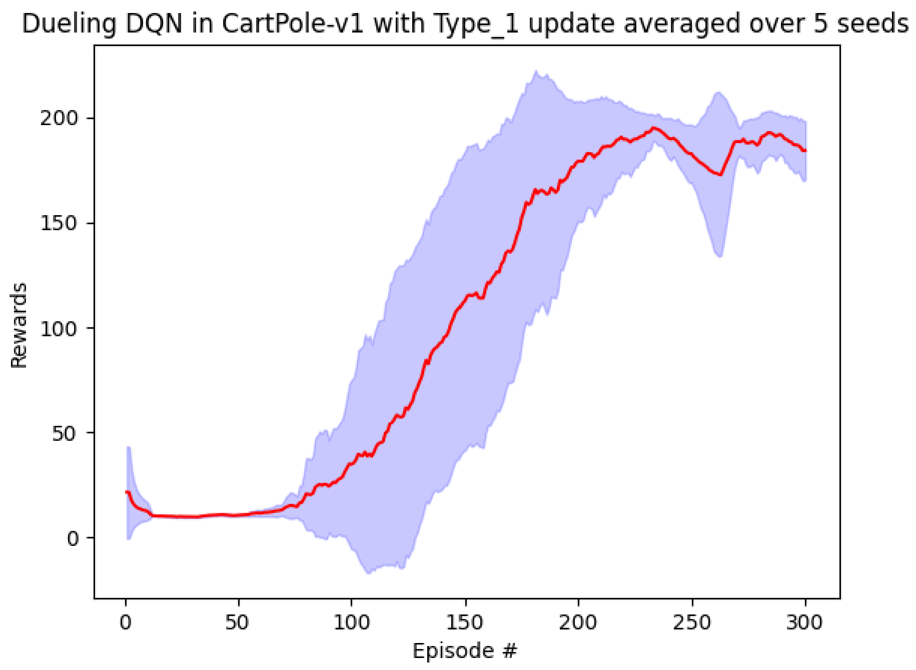
Above I have attached 3 images of code snippets below. That is my main learning function which can perform one full experiment. We run it 5 times with different initial seeds to get mean episodic rewards and variance range.

Results & Inference :

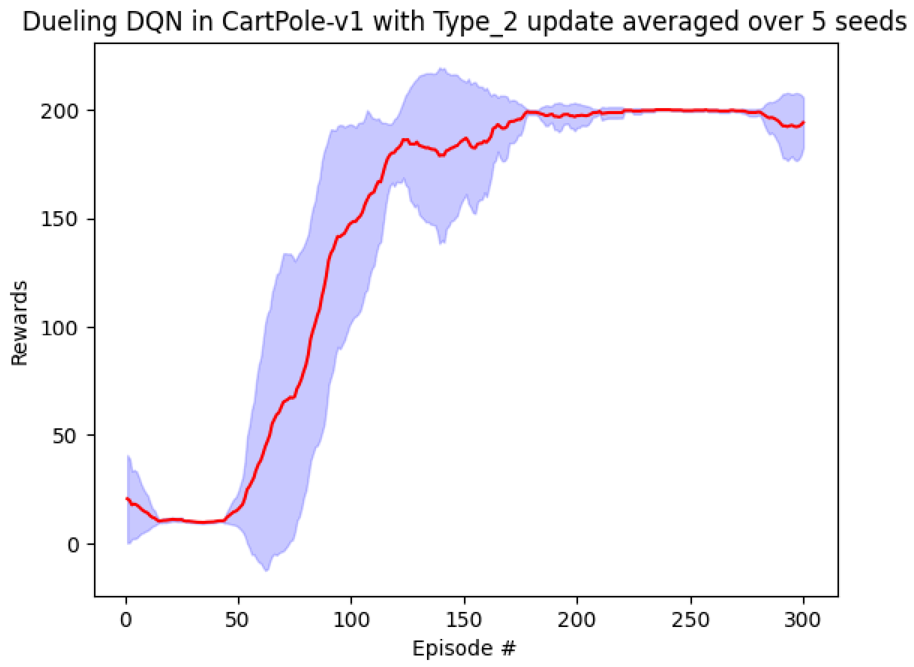
- We are using the **maximum** and **average** advantage aggregation and trying to compare the results for two different types of environments - Acrobot and Cartpole.
- In both of these environments - Acrobot and Cartpole, **Type 2** update (i.e **maximum advantage aggregation)** is performing **better than Type 1** update (i.e **mean advantage aggregation)**
- But what is interesting to see is the difference in performance of aggregation types in different types.

CartPole Environment =>

1. **Dueling DQN with Type-1 update (mean of advantage aggregation) :**

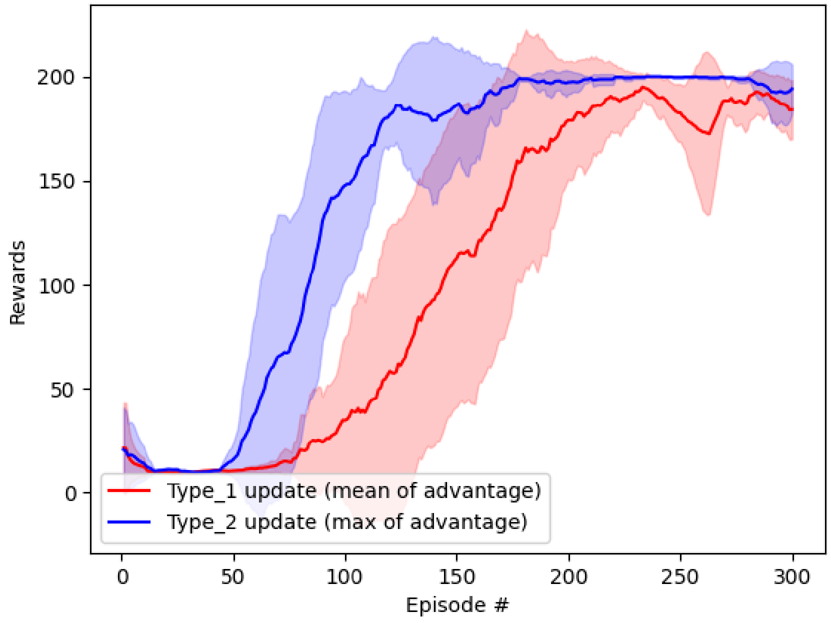


2. **Dueling DQN with Type-2 update (max of advantage aggregation) :**



3. Comparison between both update types =>

Comparitive performance of Dueling DQN in CartPole-v1 for 2 update types averaged over 5 seeds



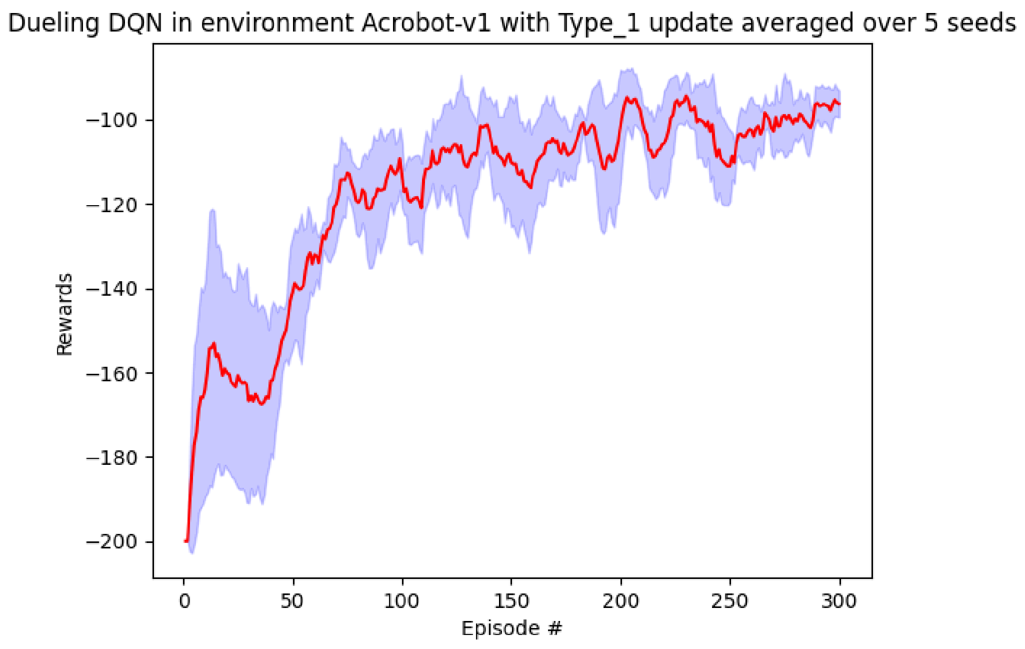
- **Type 2 (maximum)** advantage aggregation is **clearly better** than **Type 1 (average)** advantage aggregation.
- Variance for **Type 2 (maximum)** advantage aggregation is also **clearly lesser** than **Type 1 (average)** advantage aggregation. General variance range is way narrower for **Type 2 (maximum)** advantage aggregation than **Type 1 (average)** advantage aggregation. This means that across all 5 experiments, **Type 2** update achieves similar rewards and is more stable than **Type 1.**
- You can see that as episodes progress, the variance is almost zero for **Type 2 (maximum)** advantage aggregation. After around 200 episodes, this type of advantage aggregation managed to achieve a maximum reward of 200 across all 5 experiments. For **Type 1 (average)** advantage aggregation, variance range is still pretty high.
- **Type 2** update curve has a **way steeper** incline than **Type 1** update. You can see that, **Type 2** update curve started increasing at around **50** episodes and reached a maximum at around **175** episodes. **Type 1** update curve started

increasing at around **100** episodes and reached a maximum at around **250-300** episodes.

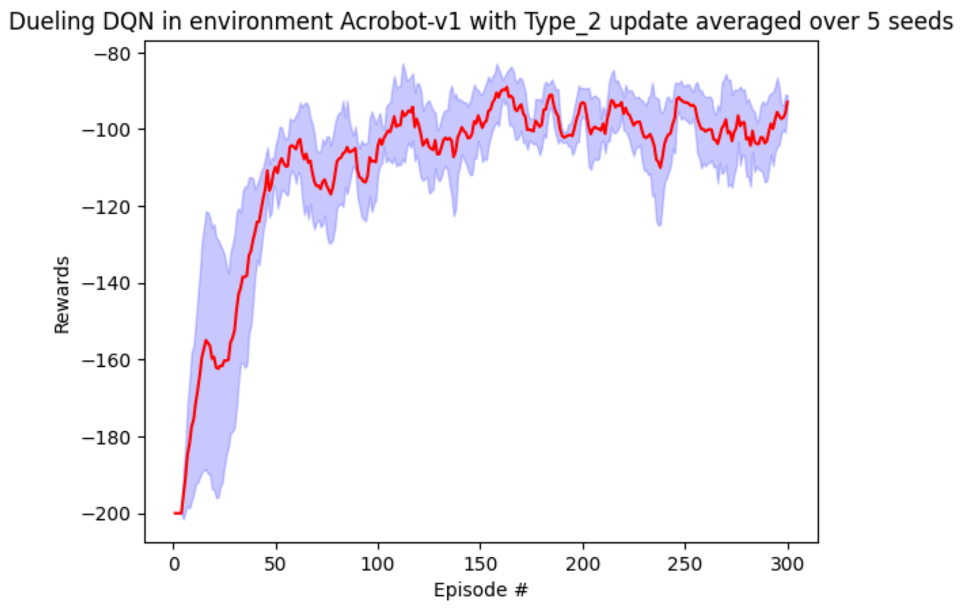
- From all this it seems like **Type 2 (maximum)** advantage aggregation is a better choice than **Type 1 (average)** advantage aggregation.
- The reason for this is most probably the **sparse rewards structure** of the Cartpole environment which prefers max advantage aggregation.
- Since successful actions (balancing the pole) are less frequent, focusing on the actions with the highest advantage (max aggregation) helps the agent learn faster and achieve the maximum reward easily across all experiments.
- In the Cartpole environment, we have binary action space and the maximum advantage aggregation helps a lot in identifying the clear best action at most points in time.

Acrobot Environment =>

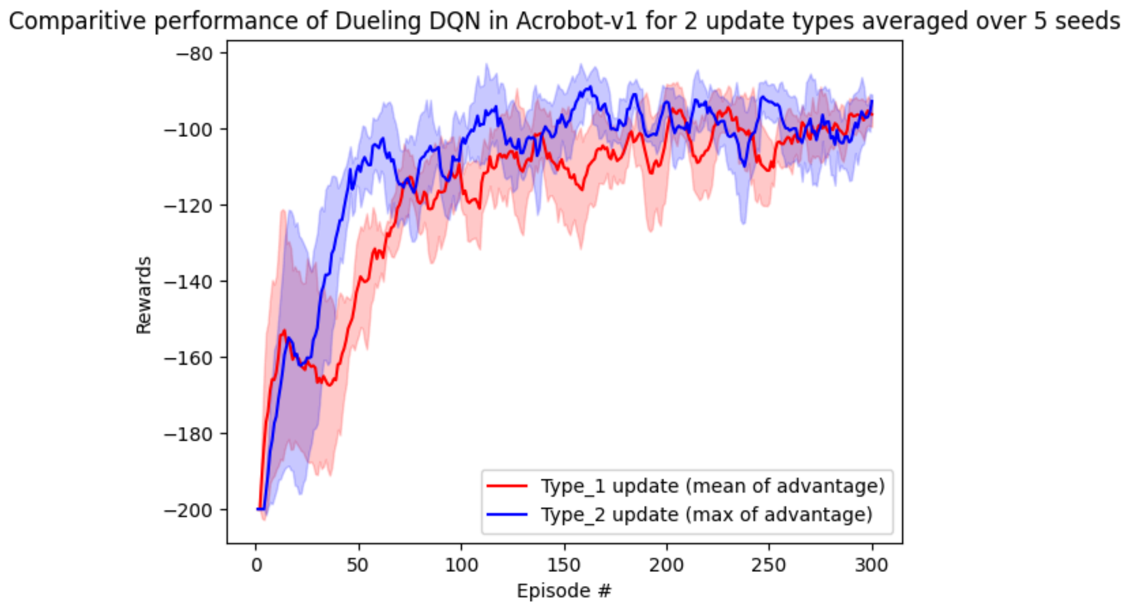
1. **Dueling DQN with Type-1 update (mean of advantage aggregation) :**



2. Dueling DQN with Type-2 update (max of advantage aggregation) :



3. Comparison between both update types =>



- Both **Type 2 (maximum)** and **Type 1 (average)** advantage aggregation are performing almost equally well.

- Initially **Type 1 or average** advantage aggregation is performing **slightly better** than **Type 2 or maximum** advantage aggregation. It also seems like at the end of episodes, **Type 1 update** is again **slightly outperforming** the **Type 2 update**.
- Initially for both update types, variance is pretty high. After **50 episodes**, the variance range decreases significantly. For both update types, the variance range is almost similar.
- Compared to the Cartpole environment, the Acrobot environment offers a more frequent or denser reward structure.
- Unlike the Cartpole environment, the agent gets rewards for various limb configurations and not just one upright position. So it might be better if we consider the advantages across all actions instead of just the maximum advantage.
- So in this case, it will be **better to use** the **Type 1 or average** advantage aggregation over **Type 2 or maximum**, because it might be a bit more stable.

GITHUB LINK =>

https://github.com/Gilgamesh60/PA2_CH21B062_CS6700.git

CS6700 - RL

MC REINFORCE results

MC REINFORCE Introduction (Ignore) =>

MC REINFORCE is one of the policy gradient algorithms. In policy gradient methods, we search directly for the optimal policy by performing a gradient ascent based on the performance of some kind of objective function. In the MC REINFORCE method, we want to increase the probability of those actions that yield the maximum returns. For this, we calculate the **estimated return i.e G_t** from an entire episode to update the policy of the agent.

This **estimated return i.e G_t** is calculated using Monte Carlo sampling. We collect a trajectory (samples of experiences) and calculate the discounted return, and use this score to increase or decrease the probability of every action taken on that trajectory. If the return is good, all actions will be "reinforced" by increasing their likelihood of being taken.

To do this we define a policy parameter θ which will be updated using MC REINFORCE algorithm. The update equation for this is given by :

1. Without Baseline :

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

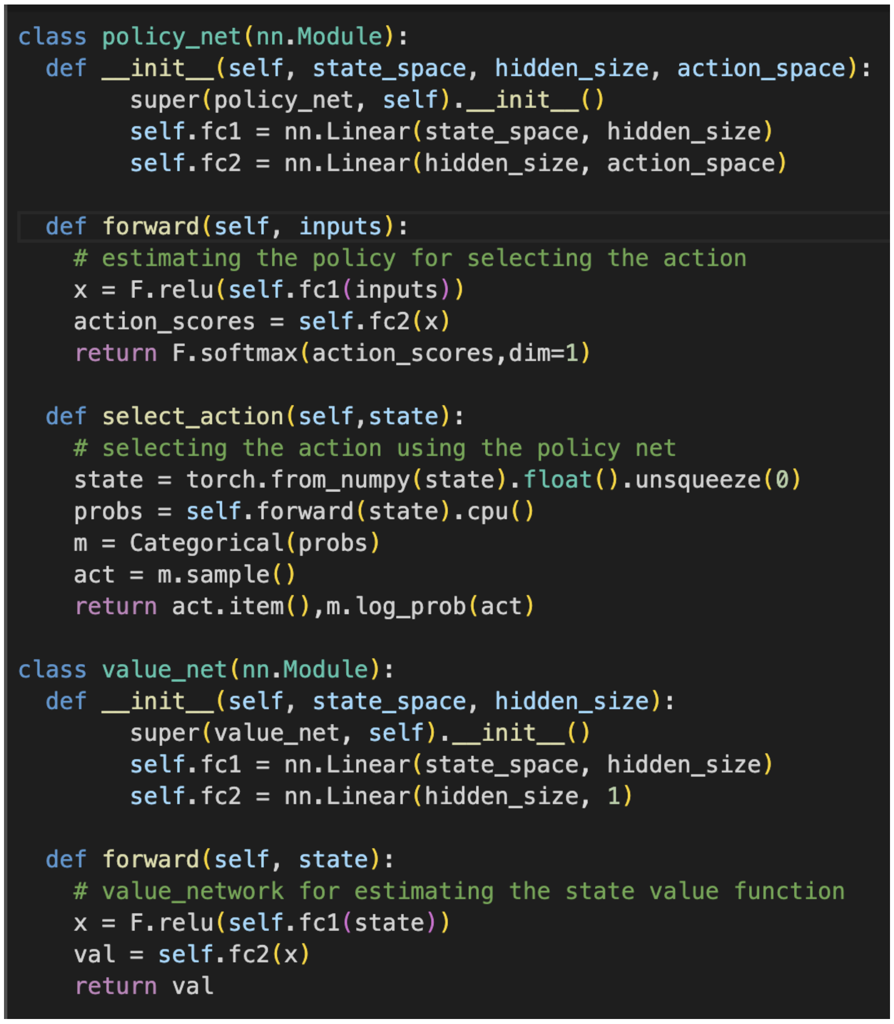
2. With Baseline :

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

With the **Baseline MC REINFORCE** method, to estimate the V (S_t; Φ) , we have to define a value network which will keep track of the estimated state value function which can be used in the update equation.

Code Snippets =>

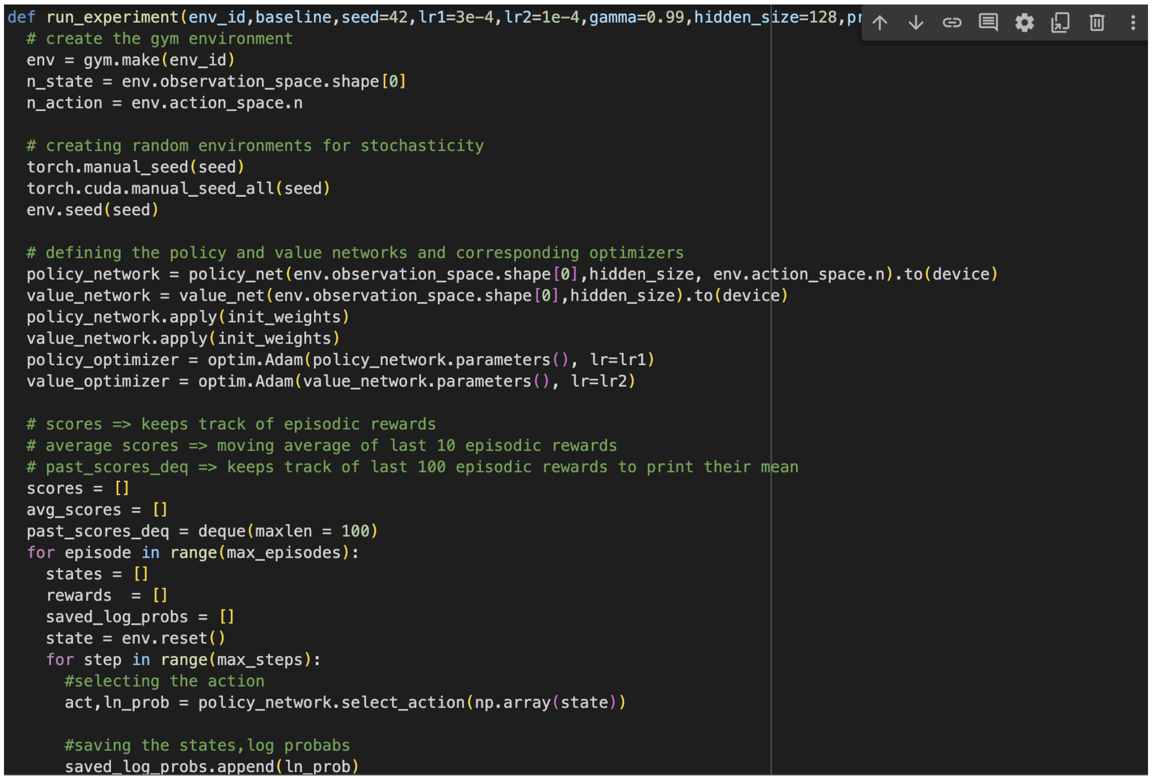
1. **Policy & Value Networks for Estimations :**



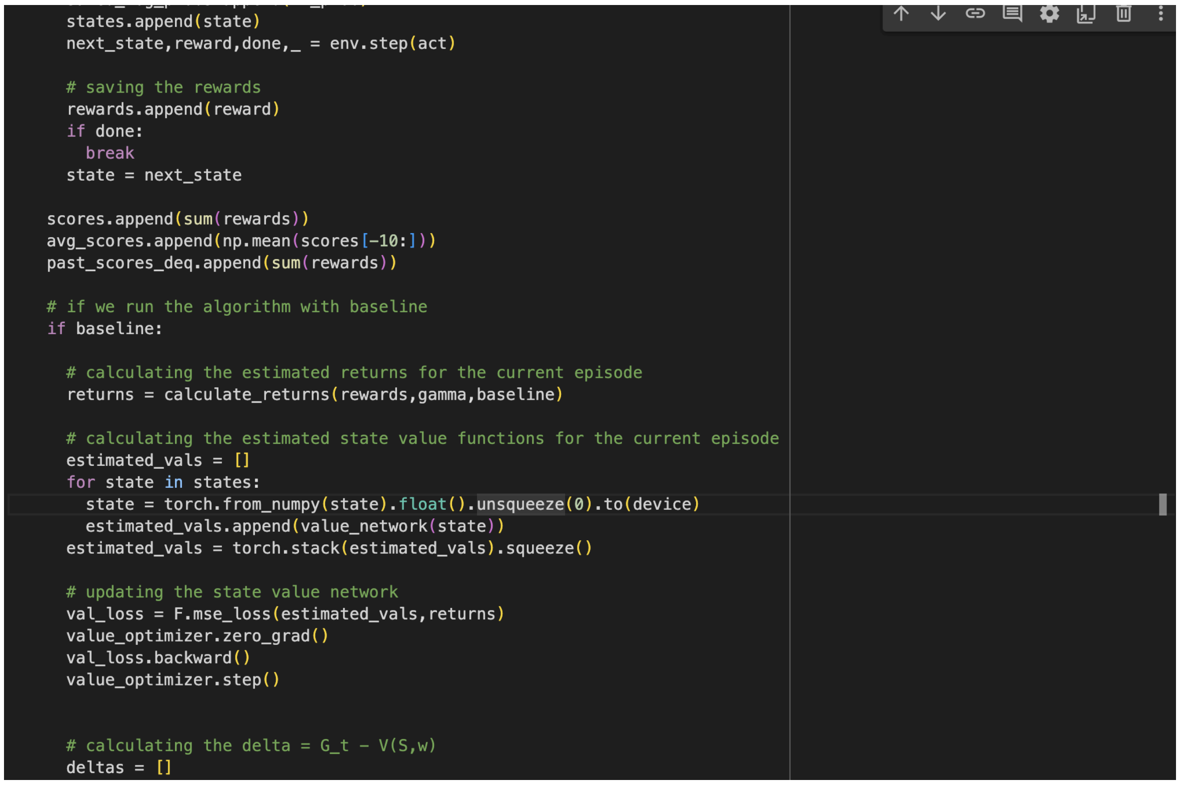
```python
class policy_net(nn.Module):
  def __init__(self, state_space, hidden_size, action_space):
        super(policy_net, self).__init__()
        self.fc1 = nn.Linear(state_space, hidden_size)
        self.fc2 = nn.Linear(hidden_size, action_space)

  def forward(self, inputs):
    # estimating the policy for selecting the action
    x = F.relu(self.fc1(inputs))
    action_scores = self.fc2(x)
    return F.softmax(action_scores,dim=1)

  def select_action(self,state):
    # selecting the action using the policy net
    state = torch.from_numpy(state).float().unsqueeze(0)
    probs = self.forward(state).cpu()
    m = Categorical(probs)
    act = m.sample()
    return act.item(),m.log_prob(act)

class value_net(nn.Module):
  def __init__(self, state_space, hidden_size):
        super(value_net, self).__init__()
        self.fc1 = nn.Linear(state_space, hidden_size)
        self.fc2 = nn.Linear(hidden_size, 1)

  def forward(self, state):
    # value_network for estimating the state value function
    x = F.relu(self.fc1(state))
    val = self.fc2(x)
    return val
```

2. **MC REINFORCE Learning Algorithm =>**

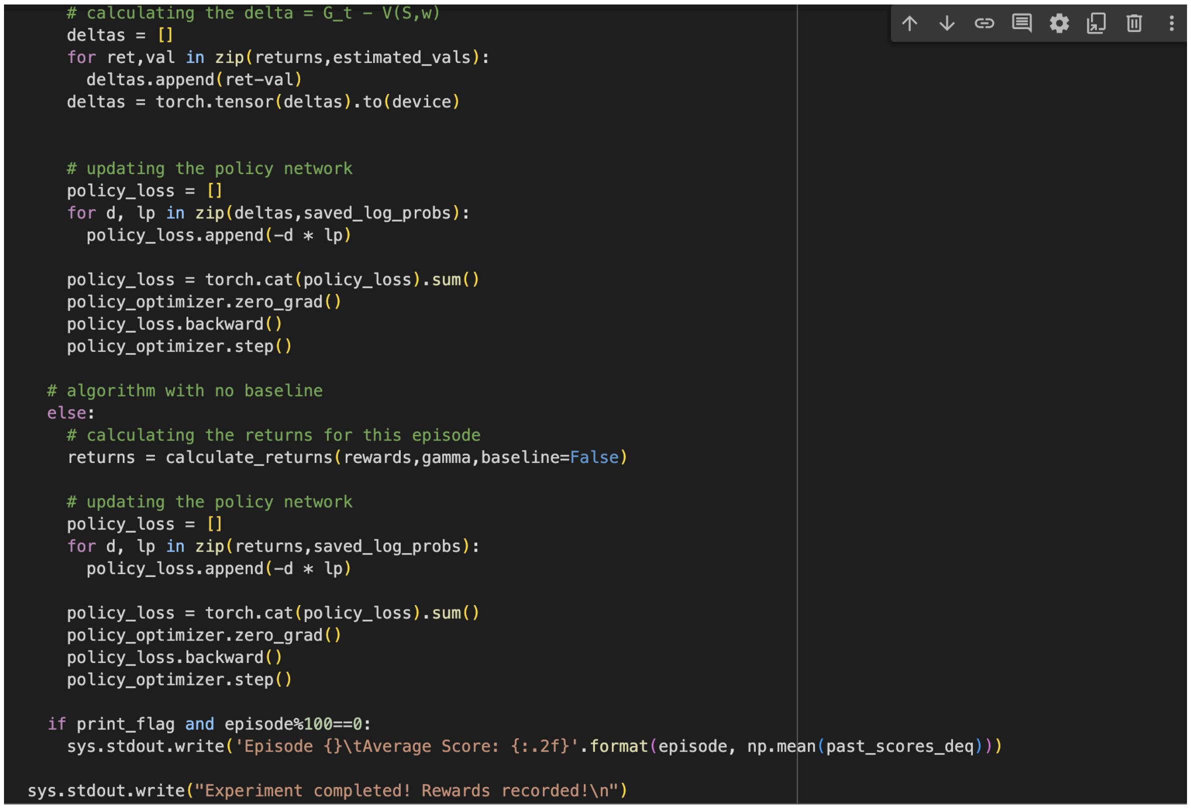
I have attached 3 images of code snippets below. That is my main learning function which can perform one full experiment. We run it 5 times with different initial seeds to get mean episodic rewards and variance range.



```
def run_experiment(env_id,baseline,seed=42,lr1=3e-4,lr2=1e-4,gamma=0.99,hidden_size=128,pr
  # create the gym environment
  env = gym.make(env_id)
  n_state = env.observation_space.shape[0]
  n_action = env.action_space.n

  # creating random environments for stochasticity
  torch.manual_seed(seed)
  torch.cuda.manual_seed_all(seed)
  env.seed(seed)

  # defining the policy and value networks and corresponding optimizers
  policy_network = policy_net(env.observation_space.shape[0],hidden_size, env.action_space.n).to(device)
  value_network = value_net(env.observation_space.shape[0],hidden_size).to(device)
  policy_network.apply(init_weights)
  value_network.apply(init_weights)
  policy_optimizer = optim.Adam(policy_network.parameters(), lr=lr1)
  value_optimizer = optim.Adam(value_network.parameters(), lr=lr2)

  # scores => keeps track of episodic rewards
  # average scores => moving average of last 10 episodic rewards
  # past_scores_deq => keeps track of last 100 episodic rewards to print their mean
  scores = []
  avg_scores = []
  past_scores_deq = deque(maxlen = 100)
  for episode in range(max_episodes):
    states = []
    rewards  = []
    saved_log_probs = []
    state = env.reset()
    for step in range(max_steps):
      #selecting the action
      act,ln_prob = policy_network.select_action(np.array(state))

      #saving the states,log probabs
      saved_log_probs.append(ln_prob)
```



```
      states.append(state)
      next_state,reward,done,_ = env.step(act)

      # saving the rewards
      rewards.append(reward)
      if done:
        break
      state = next_state

    scores.append(sum(rewards))
    avg_scores.append(np.mean(scores[-10:]))
    past_scores_deq.append(sum(rewards))

    # if we run the algorithm with baseline
    if baseline:

      # calculating the estimated returns for the current episode
      returns = calculate_returns(rewards,gamma,baseline)

      # calculating the estimated state value functions for the current episode
      estimated_vals = []
      for state in states:
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        estimated_vals.append(value_network(state))
      estimated_vals = torch.stack(estimated_vals).squeeze()

      # updating the state value network
      val_loss = F.mse_loss(estimated_vals,returns)
      value_optimizer.zero_grad()
      val_loss.backward()
      value_optimizer.step()


      # calculating the delta = G_t - V(S,w)
      deltas = []
```



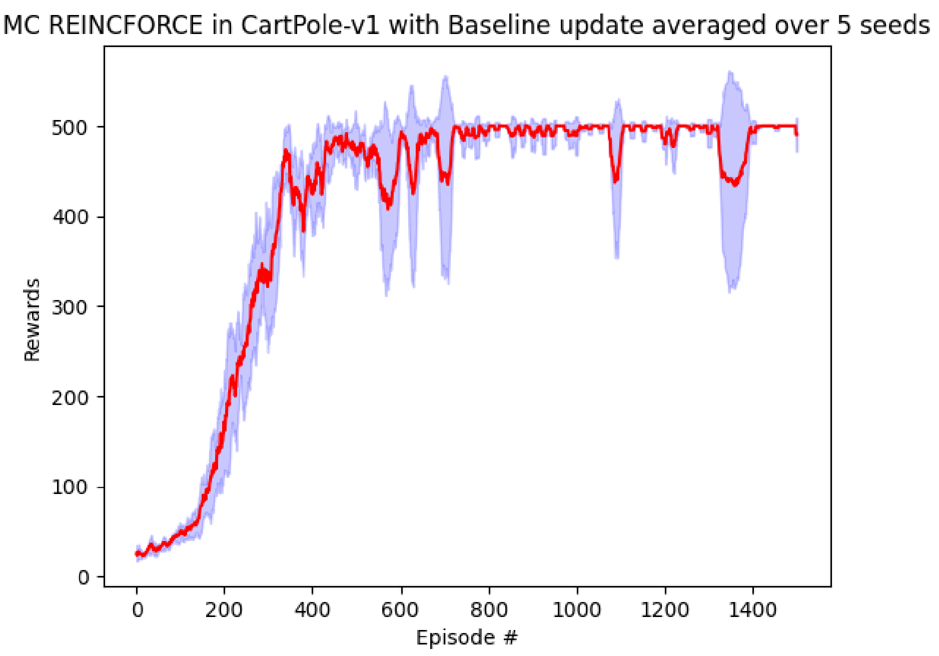
```
    # calculating the delta = G_t - V(S,w)
    deltas = []
    for ret,val in zip(returns,estimated_vals):
      deltas.append(ret-val)
    deltas = torch.tensor(deltas).to(device)


    # updating the policy network
    policy_loss = []
    for d, lp in zip(deltas,saved_log_probs):
      policy_loss.append(-d * lp)

    policy_loss = torch.cat(policy_loss).sum()
    policy_optimizer.zero_grad()
    policy_loss.backward()
    policy_optimizer.step()

  # algorithm with no baseline
  else:
    # calculating the returns for this episode
    returns = calculate_returns(rewards,gamma,baseline=False)

    # updating the policy network
    policy_loss = []
    for d, lp in zip(returns,saved_log_probs):
      policy_loss.append(-d * lp)

    policy_loss = torch.cat(policy_loss).sum()
    policy_optimizer.zero_grad()
    policy_loss.backward()
    policy_optimizer.step()

  if print_flag and episode%100==0:
    sys.stdout.write('Episode {}\tAverage Score: {:.2f}'.format(episode, np.mean(past_scores_deq)))

sys.stdout.write("Experiment completed! Rewards recorded!\n")
```

Results & Inferences =>

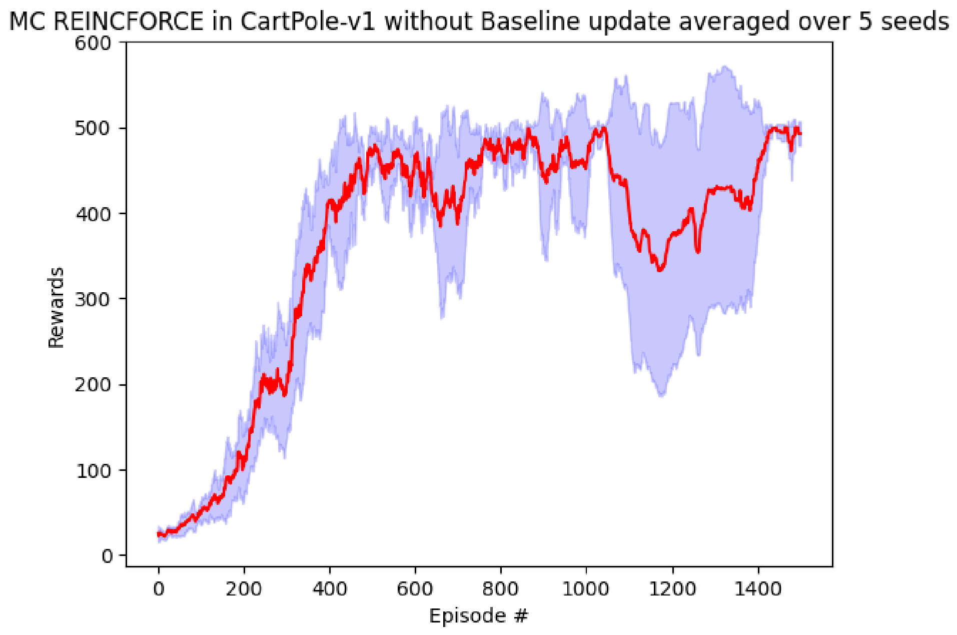
- We are trying to compare the results obtained from MC REINFORCE method with and without baseline in two different types of environments - Acrobot and Cartpole.
- In both of these environments - Acrobot and Cartpole, MC REINFORCE with baseline performed better than without baseline algorithm

CartPole Environment =>

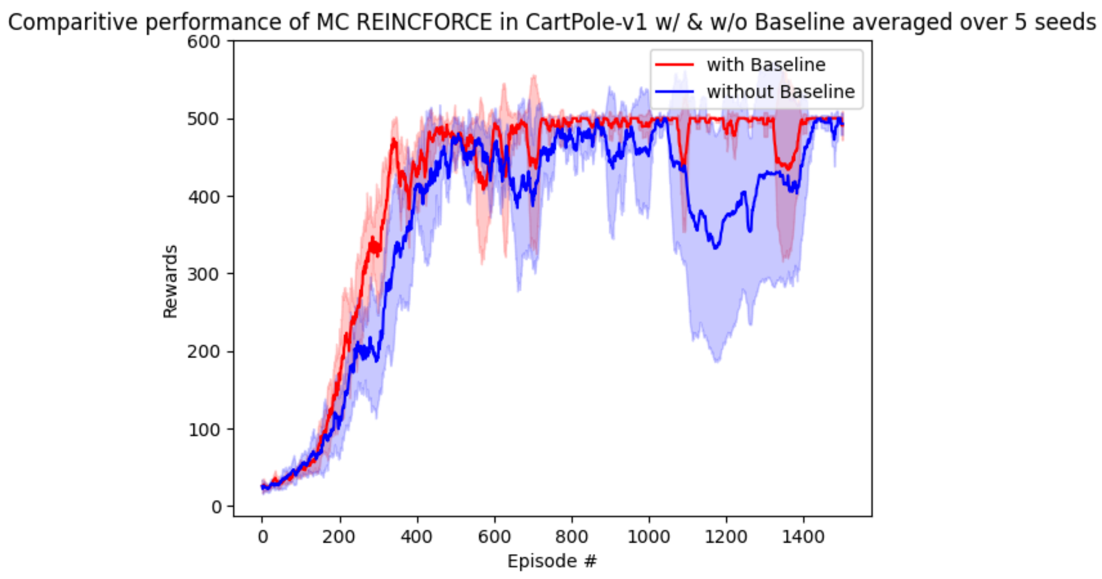
1. MC REINFORCE with baseline =>



2. MC REINFORCE without baseline =>



3. **Comparative results between two updates =>**



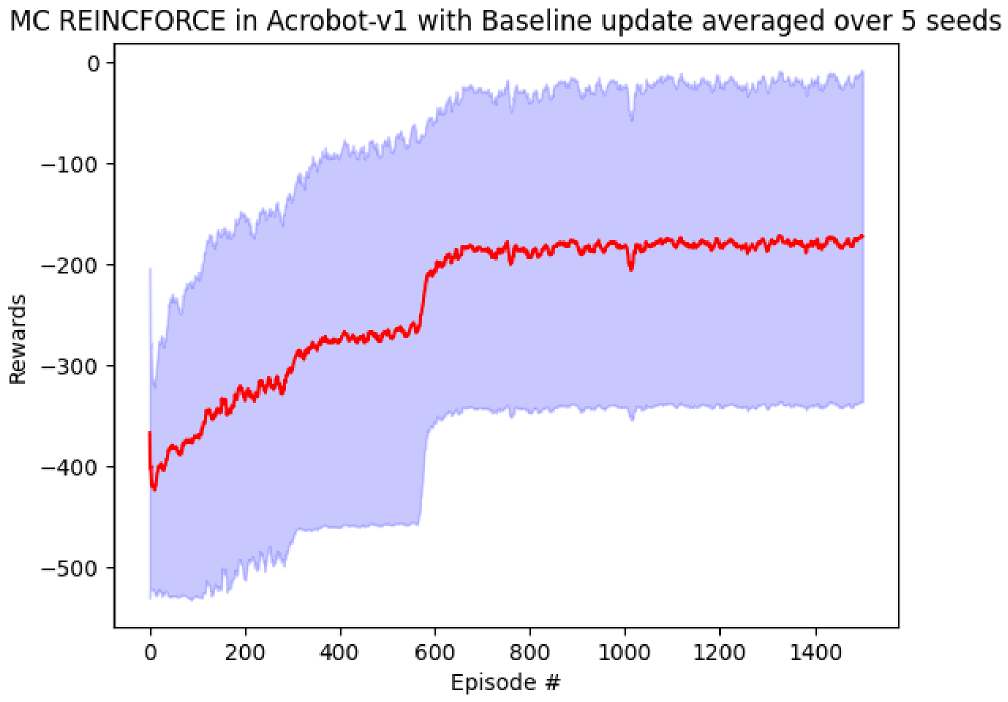
- As you can see from the graph, MC REINFORCE performs better with baseline than without baseline.
- Main goal of introducing baseline in the original MC REINFORCE method is to reduce variation of gradient estimation while keeping the bias constant.
- **Mean Episodic Rewards** => The dark red and blue lines represent the mean episodic rewards over 1500 episodes. For both types of updates, it starts low but increases as learning progresses.REINFORCE with baseline algorithm manages to get a reward of around 500. REINFORCE without baseline algorithm achieves a slightly lesser reward.
- **Variance** => The light red and blue shading represents the variance. You can see from the variation range MC REINFORCE with baseline algorithm is much more stable than without baseline algorithm. Also MC REINFORCE without baseline is much more prone to instability because it suddenly dips a lot at around 1200 episodes. On the other hand, as the episodes progress the REINFORCE with baseline algorithm manages to achieve almost a steady reward of 500.
- Main problem with no baseline algorithm is that we will be directly using the estimated returns for policy updates. So without any baseline, the agent ends

up increasing the probabilities of every action in the trajectory . It will just reward "bad" actions less than "good" actions.

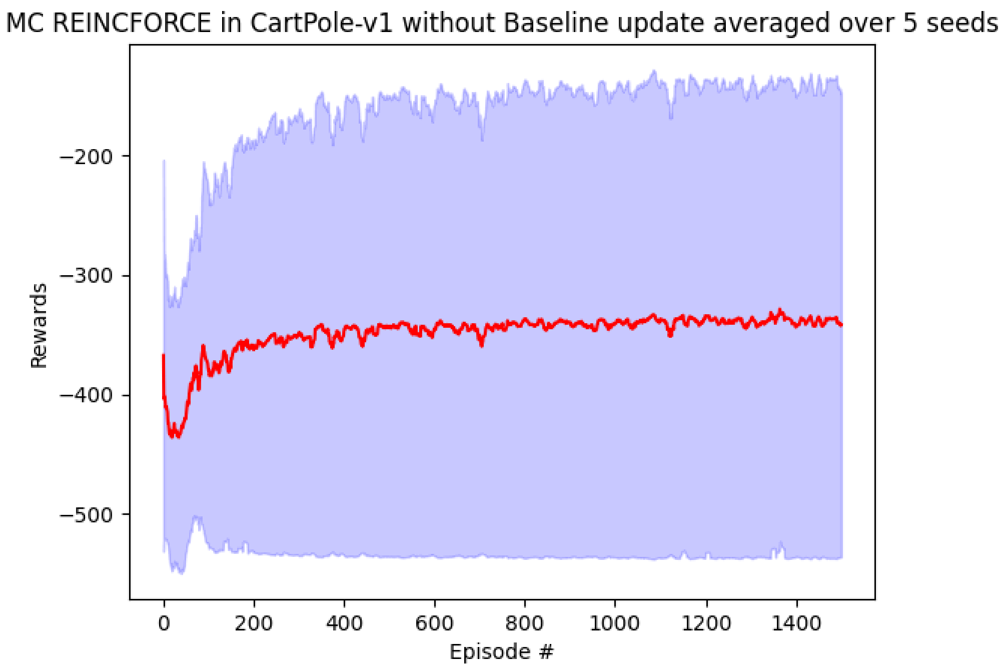
- With proper baseline, an agent punishes actions with lower estimated returns and rewards actions with higher estimated returns. So the agent can properly avoid actions with lower estimated returns.

Acrobot Environment =>

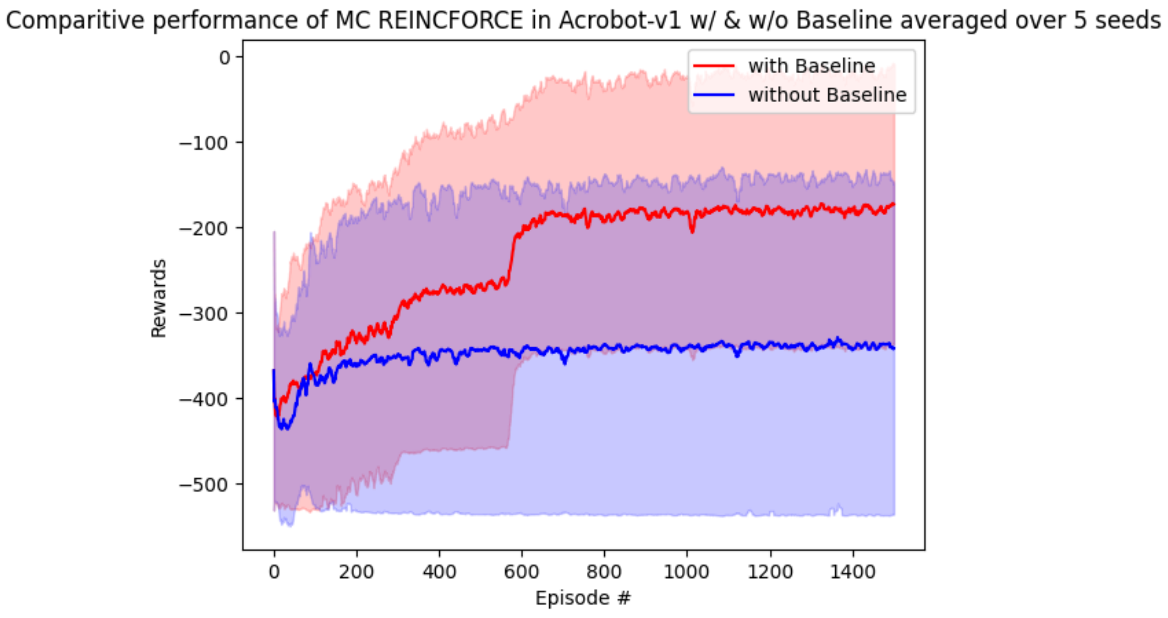
1. **MC REINFORCE with baseline =>**



2. MC REINFORCE without algorithm =>



3. Comparative performance for both updates =>



- As you can see from the graph, MC REINFORCE performs better with baseline than without baseline.
- **Mean Episodic Rewards** => The dark red and blue lines represent the mean episodic rewards over 1500 episodes. For REINFORCE with baseline, we can see an increase in the mean episodic rewards from -400 to around -200. But for the REINFORCE without baseline, mean episodic rewards doesn't really seem to change much at all. At the start there is a sharp decline in rewards after which it increases and rises slightly above its initial value.
- **Variance** => The light red and blue shading represents the variance. It is fairly clear that REINFORCE without baseline is pretty unstable. At the end of rewards variation range is pretty large i.e lesser than -500 to slightly greater than -200. Variation range is getting slightly broader as episodes progresses. In case of REINFORCE with baseline, variation range is still fairly large but at least it's better than REINFORCE without baseline. At the end of episodes, the variation range for REINFORCE with baseline is from around -350 to -100.

GITHUB LINK =>

https://github.com/Gilgamesh60/PA2_CH21B062_CS6700.git