Reinforcement Learning : Programming Assignment 2

Bhavik More CH22M009 and Rishabh Vashistha AE23S040

9th April 2024

1 Code Snippets

Github Code Repo: https://github.com/rishi1906/RL_PA_2.git

```python

# Define the Dueling DQN model
class DuelingDQN(nn.Module):
    def __init__(self, input_size, output_size, fc1_units, fc2_units, seed):
        super(DuelingDQN, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(input_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.advantage = nn.Linear(fc2_units, output_size)
        self.value = nn.Linear(fc2_units, 1)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        advantage = self.advantage(x)
        value = self.value(x)
        q_values = value + (advantage - advantage.mean(dim=-1, keepdim=True))
        return q_values

```

Figure 1: Dueling DQN Type 1

```python

# Define the Dueling DQN model
class DuelingDQN(nn.Module):
    def __init__(self, input_size, output_size, fc1_units, fc2_units, seed):
        super(DuelingDQN, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(input_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.advantage = nn.Linear(fc2_units, output_size)
        self.value = nn.Linear(fc2_units, 1)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        advantage = self.advantage(x)
        value = self.value(x)
        q_values = value + (advantage - advantage.max(dim=-1, keepdim=True)[0])
        return q_values

```

Figure 2: Dueling DQN Type 2

```
31
32          # Recalculate the total reward applying discounted factor
33          discounts = [gamma ** i for i in range(len(rewards) + 1)]
34          R = sum([a * b for a,b in zip(discounts, rewards)])
35
36          # Calculate the loss
37          policy_loss = []
38          for t, log_prob in enumerate(saved_log_probs):
39              # Note that we are using Gradient Ascent, not Descent. So we
40              policy_loss.append(-log_prob * R )
41          # After that, we concatenate whole policy loss in 0th dimension
42          policy_loss = torch.cat(policy_loss).sum()
43
44          # Backpropagation
45          optimizer.zero_grad()
46          policy_loss.backward()
47          optimizer.step()
```

Figure 3: Monte Carlo REINFORCE Type 1

```
17      scores_deque.append(sum(rewards))
18      scores.append(sum(rewards))
19      discounts = [gamma ** i for i in range(len(rewards) + 1)]
20      R = sum([a * b for a, b in zip(discounts, rewards)])
21
22      # Calculate the baseline
23      baseline_values = []
24      for state_in_trajectory in states:
25          state_tensor = torch.from_numpy(state_in_trajectory).float().unsqueeze(0).to(device)
26          baseline_value = baseline(state_tensor)
27          baseline_values.append(baseline_value)
28      baseline_values = torch.cat(baseline_values, dim=0).requires_grad_()
29
30      # Calculate the policy loss
31      policy_loss = []
32      for log_prob, baseline_value in zip(saved_log_probs, baseline_values):
33          policy_loss.append(-(log_prob * (R - baseline_value)))
34      policy_loss = torch.cat(policy_loss).sum()
35
36      # Update the policy
37      optimizer.zero_grad()
38      policy_loss.backward(retain_graph=True)  # Retain the computation graph
39      optimizer.step()
40
41      # Update the baseline
42      baseline_optimizer.zero_grad()
43      baseline_loss = ((baseline_values - R) ** 2).mean()
44      baseline_loss.backward()
45      baseline_optimizer.step()
```

Figure 4: Monte Carlo REINFORCE Type 2

```
47          episode_list_epsgrdy.append(i_episode)
48          wandb.log({'average_score': average_score})
49          wandb.log({'average_regret': average_regret})
50          wandb.log({"cummulative_regret": cumulative_regret})
51
52          print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, average_score), end="")
53
54          if i_episode % 100 == 0:
55              print('\rEpisode {}\tAverage Score: {:.2f}\tRegret: {:.2f}'.format(i_episode, average_score, regret))
56
57          if np.mean(scores_window)>=195.0:
58              print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))
59              break
60          eps = max(eps_end, eps_decay*eps)
61
62      return episode_list_epsgrdy, average_scores_epsgrdy, average_regret_epsgrdy
```

Figure 5: CartPole-V1 Termination Condition

```
47          episode_list_epsgrdy.append(i_episode)
48          wandb.log({'average_score': average_score})
49          wandb.log({'average_regret': average_regret})
50          wandb.log({"cummulative_regret": cumulative_regret})
51
52          print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, average_score), end="")
53
54          if i_episode % 100 == 0:
55              print('\rEpisode {}\tAverage Score: {:.2f}\tRegret: {:.2f}'.format(i_episode, average_score, regret))
56
57          if i_episode % 100 == 0 and np.mean(scores_window) >= -100:
58              wandb.log({"episode_no": i_episode})
59              print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))
60              break
61          eps = max(eps_end, eps_decay*eps)
62
63      return episode_list_epsgrdy, average_scores_epsgrdy, average_regret_epsgrdy
```

Figure 6: Acrobot-V1 Termination Condition

```
1   # # Average over 5 exeriments
2
3   # average_scores, cumm_regret = [], []
4   num_expts = 5
5   for i in range(num_expts):
6       print("Experiment: %d" % (i + 1))
7       # episode_average_scores, episode_cumm_regret =  wandb.agent(sweep_id, function=AvgOverExperiments, count=1)
8       wandb.agent(sweep_id, function=AvgOverExperiments, count=1)
9       # average_scores.append(episode_average_scores)
10      # cumm_regret.append(episode_cumm_regret)
11  wandb.finish()
```

Figure 7: Code for running average over 5 Experiments

```python
    def learn(self, experiences):
        states, actions, rewards, next_states, dones = experiences
        Q_targets_next = self.target_network(next_states).detach().max(1)[0].unsqueeze(1)
        Q_targets = rewards + (self.gamma * Q_targets_next * (1 - dones))
        Q_expected = self.q_network(states).gather(1, actions)
        loss = F.mse_loss(Q_expected, Q_targets)
        # wandb.log({'train loss': loss})
        self.optimizer.zero_grad()
        loss.backward()
        for param in self.q_network.parameters():
            param.grad.data.clamp_(-1, 1)  # Gradient clipping
        self.optimizer.step()

```

Figure 8: Dueling DQN Agent Learn Method

```python

for i_episode in range(1, n_episodes+1):
    state = env.reset()
    score = 0
    regret = 0  # Initialize regret for this episode
    # cumulative_regret = 0
    for t in range(max_t):
        action = agent.act(state, eps)
        next_state, reward, done, _ = env.step(action)
        optimal_action = np.argmax(agent.q_network(torch.from_numpy(state).float().unsqueeze(0)).cpu().data.numpy())
        optimal_reward = env.step(optimal_action)[1]  # Get the reward for the optimal action
        regret += optimal_reward - reward  # Calculate regret for this time step
        agent.step(state, action, reward, next_state, done)
        state = next_state
        score += reward

        if done:
            break

```

Figure 9: Dueling DQN Regret Calculation

```python

class Policy(nn.Module):
    def __init__(self, state_size=4, action_size=2, hidden_size=32):
        super(Policy, self).__init__()
        self.fc1 = nn.Linear(state_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, action_size)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = self.fc2(x)
        # we just consider 1 dimensional probability of action
        return F.softmax(x, dim=1)

    def act(self, state):
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        probs = self.forward(state).cpu()
        model = Categorical(probs)
        action = model.sample()
        return action.item(), model.log_prob(action)

```

Figure 10: Monte Carlo REINFORCE Policy Network

```python
class Baseline(nn.Module):
    def __init__(self, state_size, hidden_size=32):
        super(Baseline, self).__init__()
        self.fc1 = nn.Linear(state_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, 1)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = self.fc2(x)
        return x
```

Figure 11: Monte Carlo REINFORCE Baseline

```python
# Sweep configuration
sweep_config = {
    "method": "grid",
    "metric": {"goal": "minimize", "name": "cumulative_regret"},
    "parameters": {
        'state_size': {
            'values': [state_shape]
        },
        'action_size': {
            'values': [action_shape]
        },
        'BUFFER_SIZE': {
            'values': [int(1e5)]
            # 'values': [int(1e4), int(1e5), int(1e6)]
        },
        'BATCH_SIZE': {
            'values': [32, 64]
            # 'values': [32, 64, 128, 256]
        },
        'LR': {
            'values': [0.001, 0.0001]
            # 'values': [0.1, 0.01, 0.001, 0.0001]
        },
        'UPDATE_EVERY': {
            'values': [10, 20]
            # 'values': [4, 6, 10, 20]
        },
        'fc1_units': {
            # 'values': [128]
            'values': [64, 128]
            # 'values': [64, 128, 256]
        },
        'fc2_units': {
            # 'values': [64]
            'values': [64, 128]
            # 'values': [64, 128, 256]
        },
        'eps_start': {
            'values': [1]
        },
        'eps_end': {
            'values': [0.01]
            # 'values': [0.01, 0.05, 0.1]
        },
        'eps_decay': {
            'values': [0.99, 0.995]
            # 'values': [0.9, 0.95, 0.99, 0.995, 0.999]
        },
        'gamma': {
            'values': [0.99]
        },
        'n_episodes': {
            'values': [2000]
            # 'values': [1000, 2000, 5000]
        },
        'max_t': {
            'values': [500, 1000]
            # 'values': [500, 1000, 2000]
        },
    }
}
```

(a) Dueling DDQN Sweep Config

```python
sweep_config = {
    'method': 'grid',
    'metric': {
      'name': 'Average Reward',
      'goal': 'maximize'
    },
    'parameters': {
        'state_size': {
            'values': [state_shape]
        },
        'action_size': {
            'values': [action_shape]
        },
        'hidden_size': {
            'values': [32,64,128]
        },
        'max_t': {
            'values': [500, 1000]
        },
        'lr': {
            'values': [1e-2, 1e-3]
        },
        'baseline_lr': {
            'values': [1e-2, 1e-3]
        },
        'gamma': {
            'values': [0.99]
        },
        'alpha': {
            'values': [0.001, 0.0001]
        },
        'n_episodes': {
            'values': [2000]
        },
        "print_every": {
            'values': [100]
        }
    }
}
```

(b) Monte Carlo REINOFRCE Sweep Config

Figure 12: Sweep Configuration for wandb for hyper-parameter values

2 Experiment Setup

Each simulation followed a two-step process aimed at narrowing down the search space of hyper-parameters.

The experiments unfolded across three phases, encompassing (2 Algorithms) $\times$ (2 variants of Algorithms) $\times$ (2 environments).

To account for stochasticity, we used the average of 5 random seeds for each experiment/plot

During the first phase, we executed all experiments using varied ranges of hyperparameters through grid search. This phase aimed to identify and eliminate individual hyperparameters that yielded lower values of regret and/or episode counts. The number of episodes was determined according to environment-specific termination conditions, which are discussed later.

To account for stochasticity, we used the average of 5 random seeds for each experiment/plot

In the second phase, we operated within a narrowed-down hyperparameter search space derived from phase 1. This enabled us to effectively discern the impact of hyperparameters on regret and rewards and thus obtained best hyperparameters.Here their average over 5 experiments' averaged plot with mean and standard deviation over the 5 experiments were calculated, to account for stochasticity.

Finally, in the third phase, we utilized the best combination of hyperparameters identified in the second phase . For a given environment and algorithms, we plotted the optimal hyperparameters for each variant on the same graph, facilitating comparison.

3 Hyper Parameter Tuning

We followed 2 approaches to tune hyper parameters for every experiment mentioned in table (table no).

In first phase we tried to narrow down selection of important parameters.

In second phase we conducted experiments over average of 5 experiments with different set of hyper parameter from the set of chosen important parameters from first approach. We used WandB for grid search hyperparameter tuning and the plots corresponding to the hyperparameter sweeps

S. No.	**Algorithm**	**Environment**	**Algorithm variant**	**Code name in repository**	
				Phase 1	**Phase 2**
1	Dueling DQN	CartPole-v1	Type1	D1CS1	D1CS2
2	Dueling DQN	CartPole-v1	Type2	D2CS1	D2CS2
3	Dueling DQN	Acrobot	Type1	D1AS1	D1AS2
4	Dueling DQN	Acrobot-v1	Type2	D2AS1	D2AS2
5	Monte Carlo REINFORCE	CartPole-v1	Without Baselines	RWBCS1	RWBCS2
6	Monte Carlo REINFORCE	CartPole-v1	With Baselines	RBCS1	RBCS2
7	Monte Carlo REINFORCE	Acrobot-v1	Without Baselines	RWBAS1	RWBAS2
8	Monte Carlo REINFORCE	Acrobot-v1	With Baselines	RBAS1	RBAS1

Table 1: Set of Simulations

The termination of Cartpole-v1 environments was considered capped at 195 average reward over the episodes and the termination of Acrobot-v1 environment was considered when the average reward over the episodes reached -100 reward(From GYM Documentation)

Observations

- When size of first hidden layer is lesser than second layer, mostly solution is not found and it hits the upper limit of no episodes.
- Given other parameters are constant in grid search we observed among values of alpha performed best for value of 0.0001.
- Generally max item parameters has no significant effect on reducing the cumulative average reward. We observed larger the value of this parameters, longer it takes to converge.
- When size of hidden layers is kept same it is observed it takes longer to meet threshold criterian in Acrobot environment.
- At times we observed the average rewards suddenly dips from a very close value to exit criteria value in Cartpole problem. After this it keeps fluctuating around a very low value. It could be due to poor learning rate.

3.1 Exp. No. 1

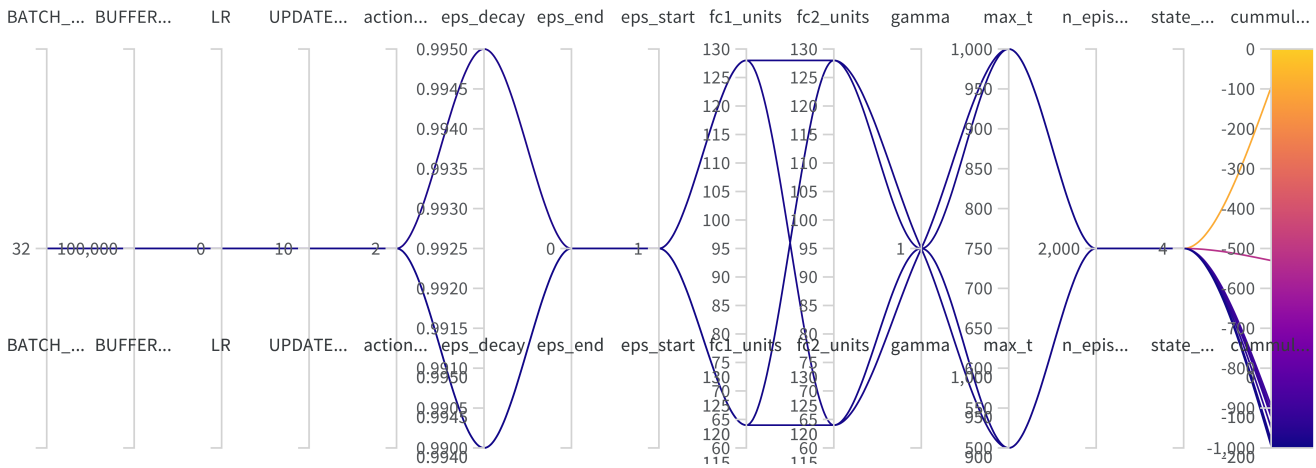

Figure 13: Wandb Sweep Profile for D1CS1

- Algorithm:Dueling DQN
- Environment: Cartpole-v1
- Algorithm Variant:Type1
- Code name :D1CS1,D1CS2

The optimal hyperparameters for this experiment were:

- BUFFER SIZE: int(1e5),
- BATCH SIZE: 64,
- Learning rate: 0.0001,
- UPDATE EVERY: 20,
- fc1 units: 128,
- fc2 units: 64,
- epsilon decay: 0.999,
- max t: 1000

3.2 Exp. No. 2

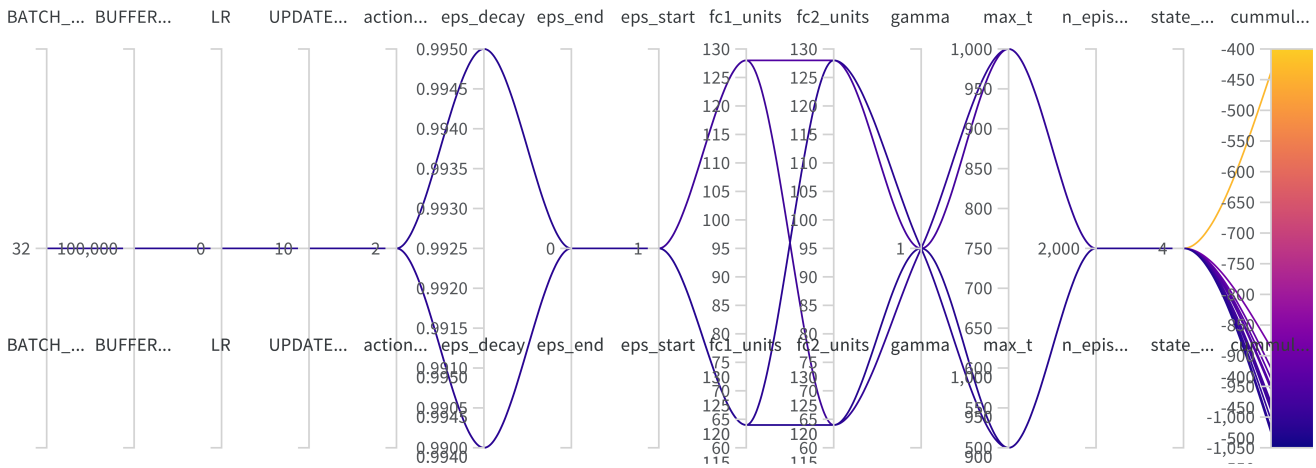

Figure 14: Wandb Sweep Profile for D2CS1

- Algorithm:Dueling DQN
- Environment: Cartpole-v1
- Algorithm Variant:Type2
- Code name :D2CS1,D2CS2

The optimal hyperparameters for this experiment were:

- BUFFER SIZE: int(1e5),
- BATCH SIZE:128,
- Learning rate: 0.001,
- UPDATE EVERY: 20,
- fc1 units: 128,
- fc2 units: 128,
- epsilon decay: 0.999,
- max t: 1000

3.3 Exp. No. 3

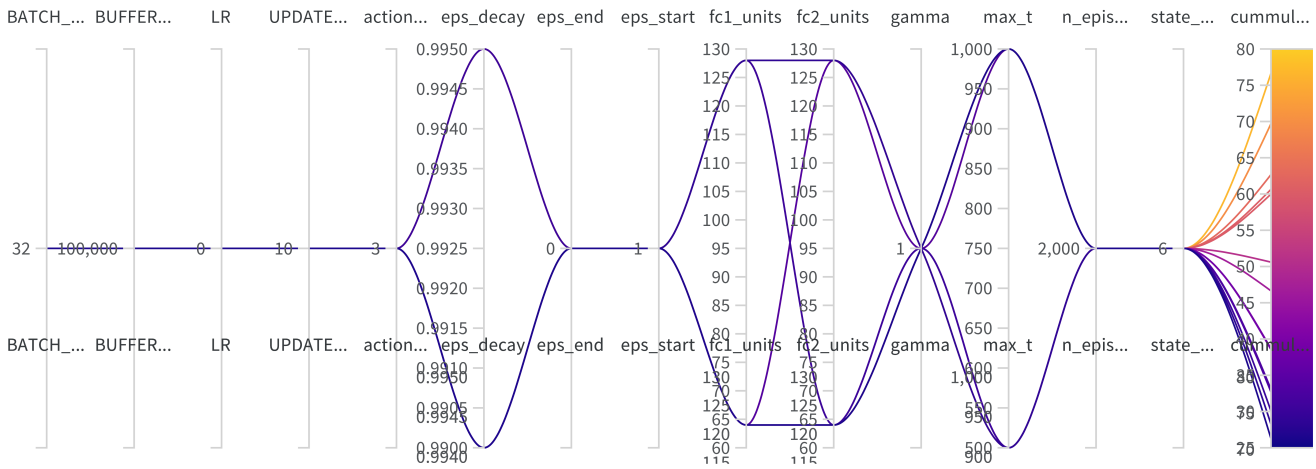

Figure 15: Wandb Sweep Profile for D1AS1

- Algorithm:Dueling DQN
- Environment: Acrobot-v1
- Algorithm Variant:Type1
- Code name :D1AS1,D1AS2

The optimal hyperparameters for this experiment were:

- BUFFER SIZE: int(1e5),
- BATCH SIZE: 64,
- Learning rate: 0.0001,
- UPDATE EVERY: 20,
- fc1 units: 128,
- fc2 units: 64,
- epsilon decay: 0.995,
- max t: 1000

3.4 Exp. No. 4

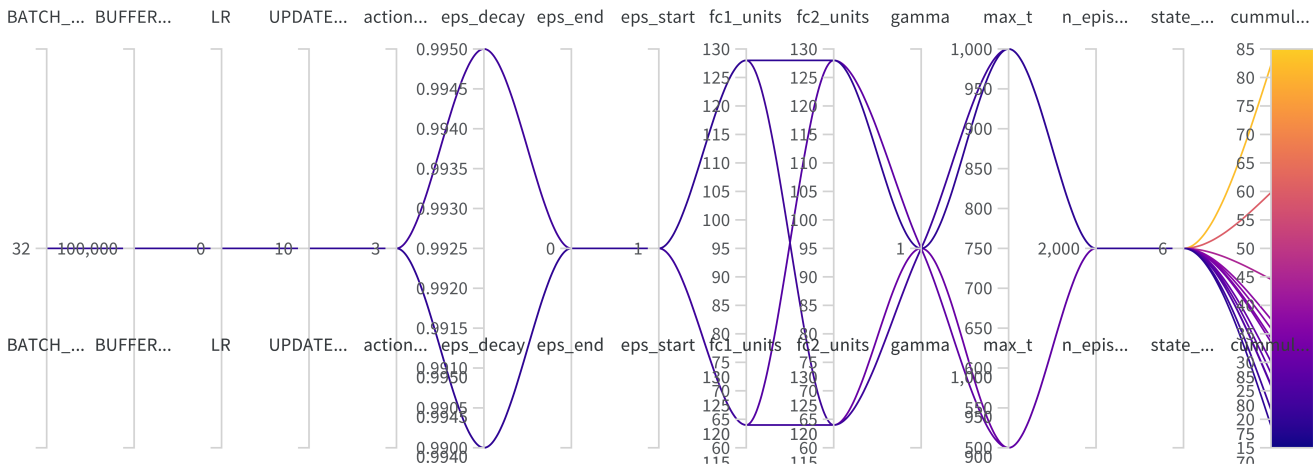

Figure 16: Wandb Sweep Profile for D2AS1

- Algorithm:Dueling DQN
- Environment: Acrobot-v1
- Algorithm Variant:Type2
- Code name :D2AS1,D2AS2

The optimal hyperparameters for this experiment were:

- BUFFER SIZE: int(1e5),
- BATCH SIZE: 64,
- Learning rate: 0.0001,
- UPDATE EVERY: 20,
- fc1 units: 128,
- fc2 units: 64,
- epsilon decay: 0.999,
- max t: 1000

3.5 Exp. No. 5

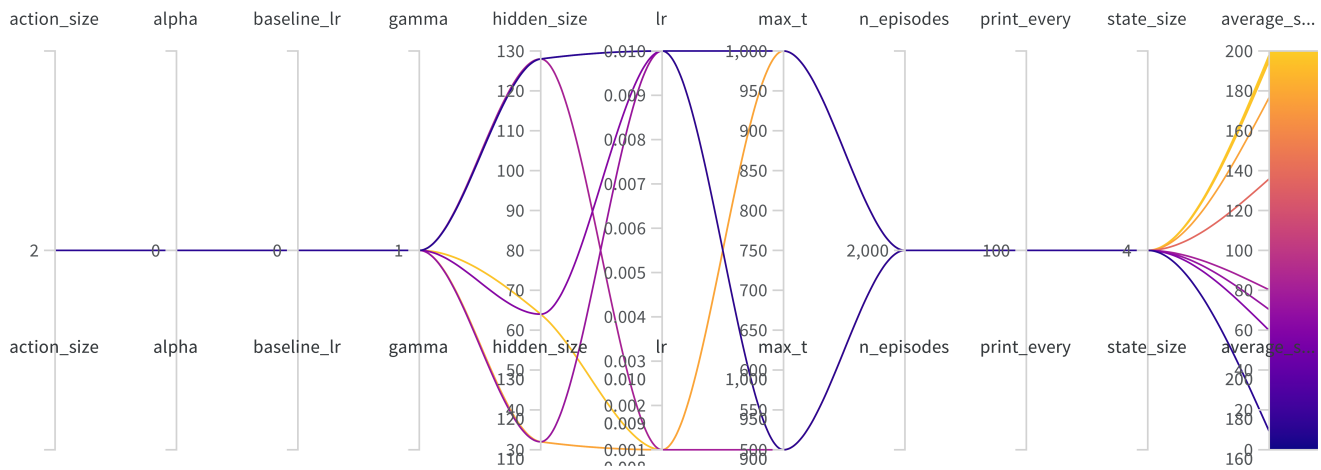

Figure 17: Wandb Sweep Profile for RWBCS1

- Algorithm:Monte Carlo REINFORCE
- Environment: Cartpole-v1
- Algorithm Variant:Without Baselines
- Code name :RWBCS1,RWBCS2

The optimal hyperparameters for this experiment were:

- BUFFER SIZE: int(1e5),
- BATCH SIZE: 64,
- Learning rate: 0.0001,
- UPDATE EVERY: 20,
- fc1 units: 128,
- fc2 units: 64,
- epsilon decay: 0.999,
- max t: 1000

3.6 Exp. No. 6

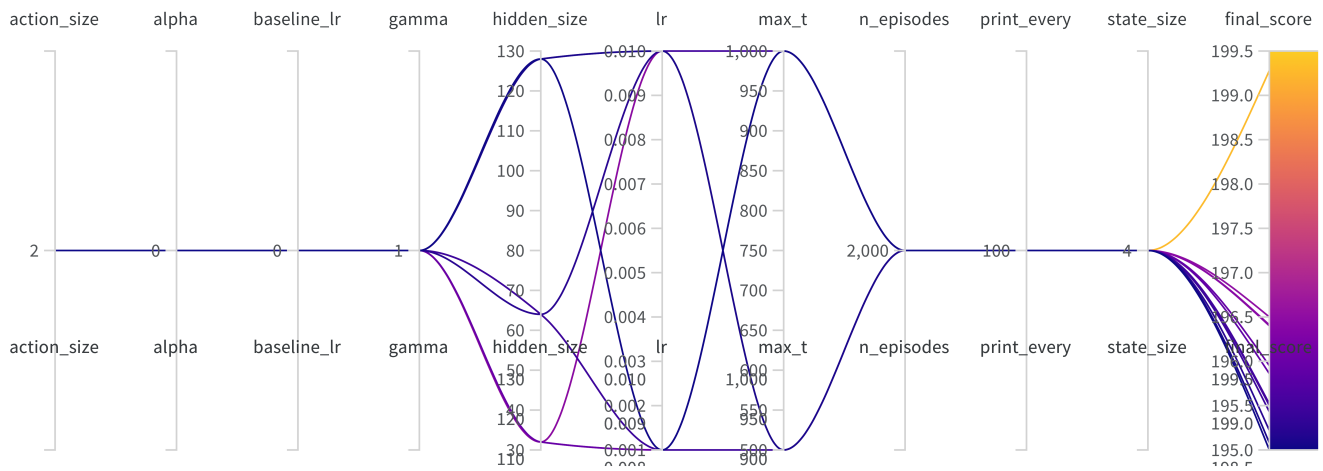

Figure 18: Wandb Sweep Profile for RBCS1

- Algorithm:Monte Carlo REINFORCE
- Environment: Cartpole-v1
- Algorithm Variant:With Baselines
- Code name :RBCS1,RBCS2

The optimal hyperparameters for this experiment were:

- BUFFER SIZE: int(1e5),
- BATCH SIZE: 32,
- Learning rate: 0.001,
- UPDATE EVERY: 20,
- fc1 units: 128,
- fc2 units: 64,
- epsilon decay: 0.995,
- max t: 1000

3.7 Exp. No. 7

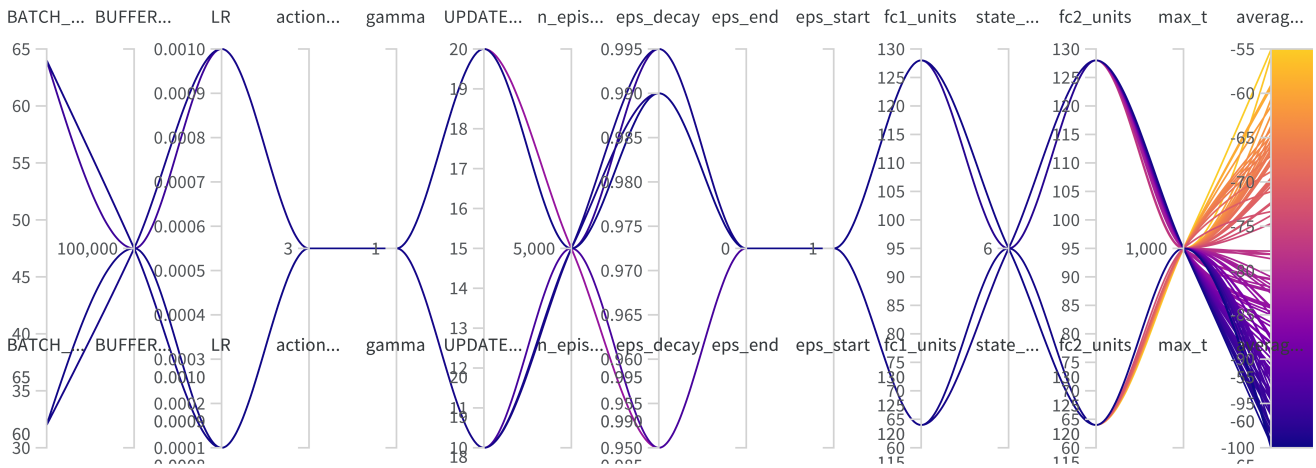

Figure 19: Wandb Sweep Profile for RWBAS1

- Algorithm:Monte Carlo REINFORCE
- Environment: Acrobot-v1
- Algorithm Variant:Without Baselines
- Code name :RWBAS1,RWBAS2

The optimal hyperparameters for this experiment were:

- BUFFER SIZE: int(1e5),
- BATCH SIZE: 32,
- Learning rate: 0.0001,
- UPDATE EVERY: 20,
- fc1 units: 128,
- fc2 units: 128,
- epsilon decay: 0.999,
- max t: 1000

3.8 Exp. No. 8

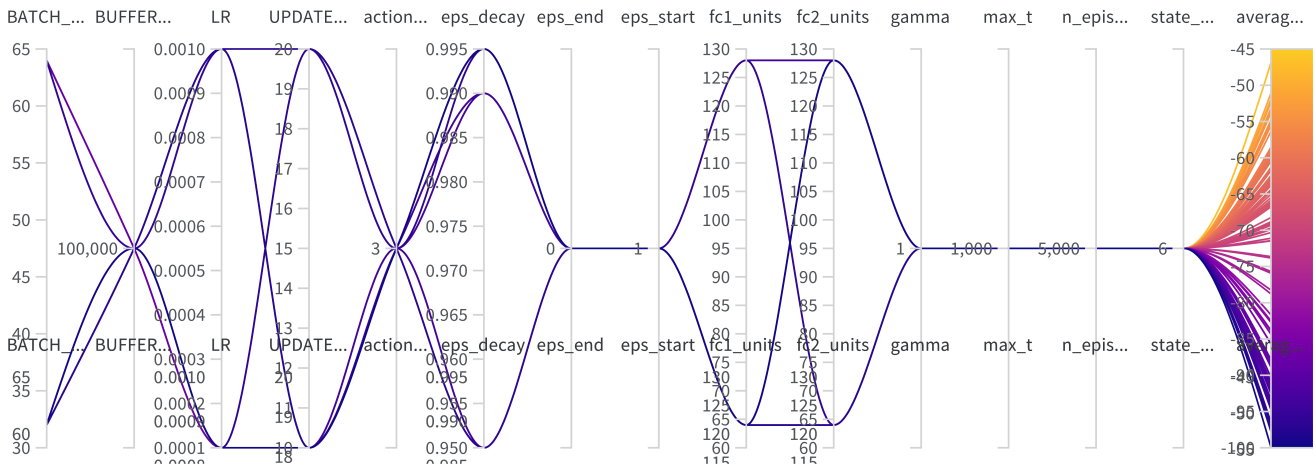

Figure 20: Wandb Sweep Profile for RBAS1

- Algorithm:Monte Carlo REINFORCE
- Environment: Acrobot-v1
- Algorithm Variant:With Baselines
- Code name :RBAS1,RBAS2

The optimal hyperparameters for this experiment were:

- BUFFER SIZE: int(1e5),
- BATCH SIZE: 64,
- Learning rate: 0.00001,
- UPDATE EVERY: 20,
- fc1 units: 128,
- fc2 units: 64,
- epsilon decay: 0.999,
- max t: 1000

4 Results

Figure 21: Average Reward Plot for Type 1 vs Type 2 for Acrobot-V1 Environment for Dueling DQN Algo

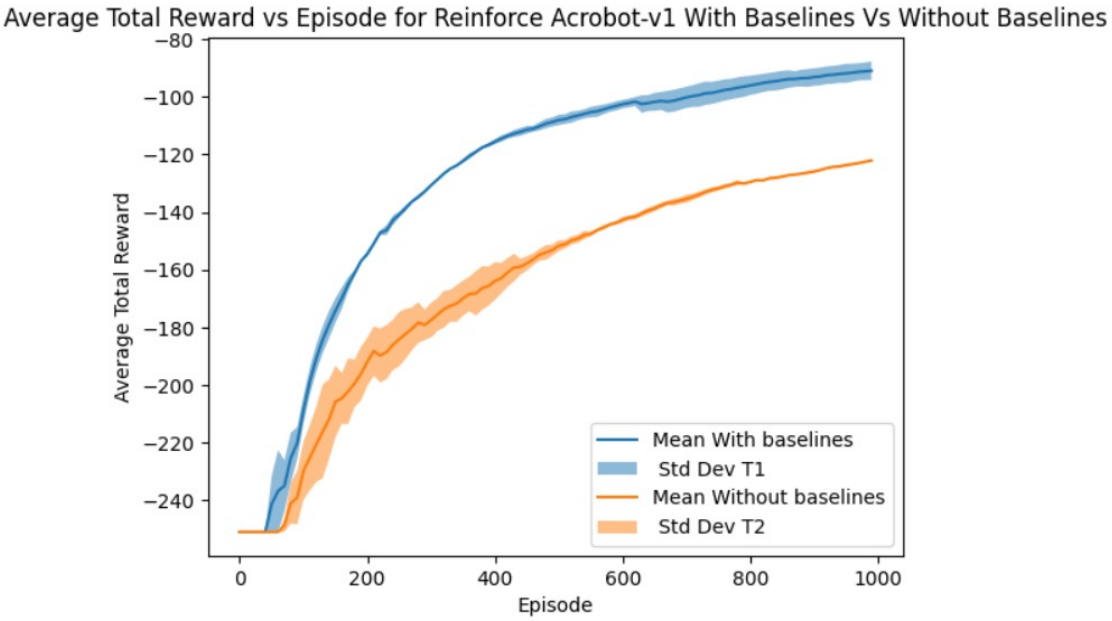

Figure 22: Average Reward Plot for Type 1 vs Type 2 for Acrobot-V1 Environment for Monte Carlo REINFORCE algo