Reinforcement Learning Assignment 3

Experiment Report

MD RASEL MAHMUD
Student ID: SL23225002

January 17, 2025

1 Introduction

This report summarizes the work completed in a reinforcement learning assignment focused on offline RL techniques. The primary goal was to evaluate the impact of different datasets on training outcomes by implementing and analyzing the Conservative Q-Learning (CQL) and standard Q-learning algorithms. The report also includes the implementation of Behavior Cloning (BC) as an additional task. Various datasets, including those with different sizes and suboptimal data, were used to assess the algorithms' performance. Key experiments were conducted to compare the algorithms, visualize the results, and record videos. This work provides insights into how offline RL methods behave with different data and training conditions. For more detailed references please look at the project directory such as code, visual results, video records so on.

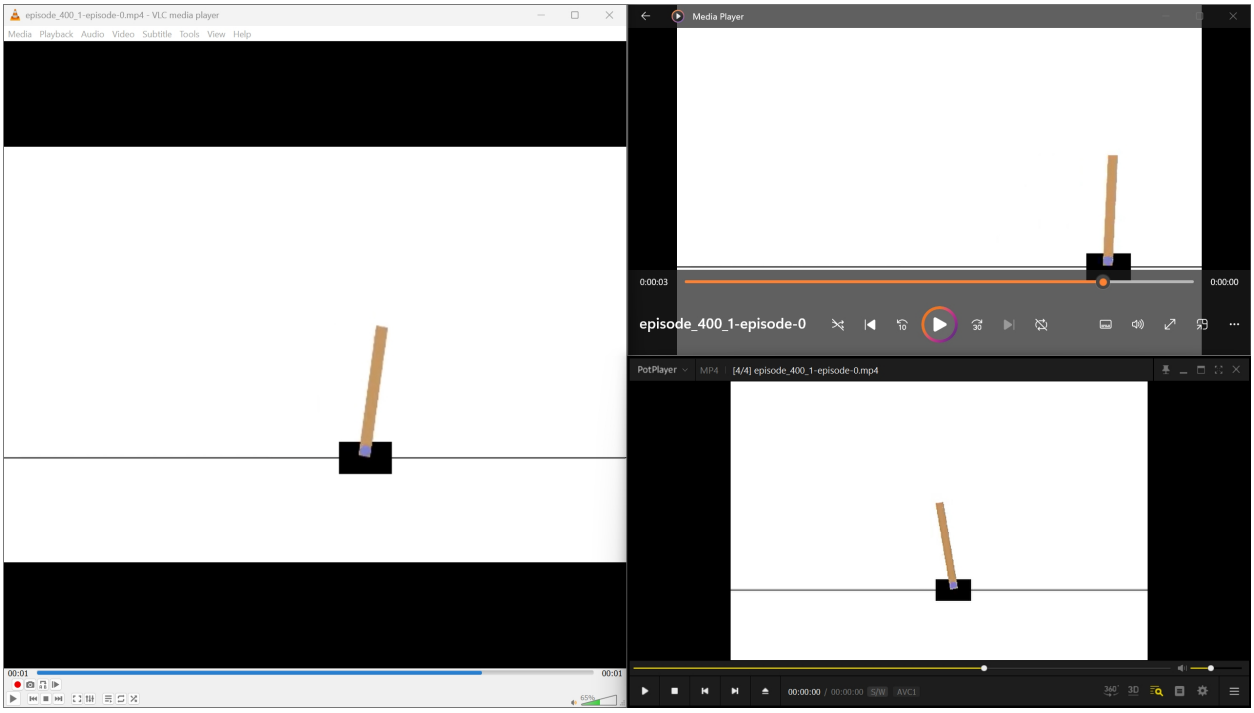


Figure 1: ALL Algorithms training video records screenshot

2 Offline Reinforcement Learning Overview

Offline reinforcement learning trains policies using a fixed, pre-collected dataset. Unlike traditional RL methods, no interactions with the environment are required, and the agent learns solely from historical data. This technique is valuable in domains where data collection is expensive or risky, such as medical decision-making or autonomous robotics.

The key challenges in offline RL include:

- **Distribution Shift**: The training data may differ from the data used during policy evaluation, leading to poor performance if the model overfits.
- **Data Quality**: The dataset may contain noise or suboptimal behaviors, which may affect training outcomes.

3 Algorithm Implementation and Training

I implemented two RL algorithms for this assignment: the standard Q-learning algorithm and the Conservative Q-Learning (CQL) algorithm. The CQL algorithm introduces a conservative term to the objective function to prevent over-fitting to out-of-distribution (OOD) actions, making the policy more robust to noisy or suboptimal data.

The training was performed using multiple datasets provided by the instructor:

- `dataset_episode_50.npz`

- `dataset_episode_150.npz`
- `dataset_episode_250.npz`
- `dataset_episode_350.npz`

Each dataset contains different numbers of training episodes (maximum 400). These datasets were used to evaluate the impact of dataset size on the training results.

3.1 CQL Implementation and Code Logic

This section explains the key steps and logic involved in implementing the **Conservative Q-Learning (CQL)** algorithm for training an offline reinforcement learning (RL) agent. The implementation is based on the training code provided in the file `CQL_Training.py`. The following sections outline the core components and their functionality.

3.1.1 Hyperparameters

The following hyperparameters are used for training the agent:

- **GAMMA** (0.99): Discount factor for future rewards.
- **TAU** (0.005): Coefficient for soft updates of target Q-networks.
- **LR_Q** (3×10^{-4}): Learning rate for Q-network.
- **LR_POLICY** (3×10^{-4}): Learning rate for the policy network.
- **ALPHA** (0.2): Temperature parameter for policy entropy.
- **BATCH_SIZE** (64): Batch size used during training.
- **MAX_EPISODES** (400): Number of episodes to train.
- **MAX_STEPS** (500): Maximum steps per episode.
- **CQL_ALPHA** (0.5): Coefficient for the CQL penalty term and it tried in different values e.g.0.5, 1.0, 2.0.
- **MIN_Q_WEIGHT** (5.0): Weight applied to the minimum Q-value penalty term and it tried in different values e.g. 5.0, 10, 20.

3.1.2 Policy and Q-Networks

The code defines two types of neural networks:

1. **Policy Network** (PolicyNetwork): A neural network that outputs log-probabilities for each discrete action. It uses two hidden layers and a ReLU activation function.

2. **Q-Networks** (QNetwork): Two Q-networks (Q1 and Q2) are used to estimate the action-value function $Q(s, a)$ for each action. Both networks have the same architecture with two hidden layers and ReLU activation.

The policy network computes the probability distribution over actions for a given state, and the Q-networks compute the action values for each action in a given state.

3.1.3 Offline Replay Buffer

The `OfflineReplayBuffer` class is used to store and sample experiences from a dataset. The dataset is loaded from a provided file (e.g., `dataset_episode_50.npz`). Each dataset consists of:

- States (`states`)
- Actions (`actions`)
- Rewards (`rewards`)
- Next states (`next_states`)
- Done flags (`dones`)

The `sample` method randomly selects a batch of experiences from the dataset, which is then used for training.

3.1.4 Training Step with CQL Loss

The core of the training process involves the following steps:

1. **Compute Target Q-values:** The target Q-values are computed using the Bellman equation. The target Q-values depend on the next state's value, which is computed using the policy network's log-probabilities and the Q-networks for the next state.
2. **CQL Loss for Q1 and Q2:** For both Q-networks (Q1 and Q2), the CQL loss is computed as a conservative regularization term:

$$\mathcal{L}_{CQL} = \alpha \left(\frac{1}{N} \sum_{i=1}^{N} Q_1(s_i, a_i) - \min_a Q_1(s_i, a) \right)$$

   where $Q_1(s, a)$ is the predicted Q-value, and $\min_a Q_1(s, a)$ is the minimum Q-value across all actions for a given state. This term penalizes actions that have higher Q-values than the minimum, encouraging conservative behavior.
3. **Q1 and Q2 Losses:** The Q1 and Q2 networks are updated to minimize the mean squared error between the predicted Q-values and the target Q-values.

4. **Policy Loss:** The policy is updated using the following loss function:

$$\mathcal{L}_{policy} = \mathbb{E}_{s\sim D}\left[\sum_a \pi(a|s)\left(\alpha \log \pi(a|s) - \min_i Q_i(s,a)\right)\right]$$

where $\pi(a|s)$ is the action probability given state s, and $\min_i Q_i(s,a)$ is the minimum of the Q-values for each action.

5. **Soft Update of Target Networks:** The target Q-networks (Q1_target and Q2_target) are updated using a soft update mechanism:

$$\theta_{target} \leftarrow \tau\theta_{current} + (1-\tau)\theta_{target}$$

where τ is a small coefficient (usually 0.005) that ensures gradual updates to the target networks.

3.1.5 Training Loop and Evaluation

The training loop consists of:

- Sampling a batch from the offline replay buffer.
- Performing a training step using the CQL loss function.
- Evaluating the policy every 10 episodes using deterministic actions for performance evaluation.
- Saving videos of agent behavior every 100 episodes.
- Plotting training metrics such as Q-losses, policy loss, and average rewards.

The evaluation of the policy is done using the `evaluate_policy` function, which runs the agent in the environment for several episodes and records the average reward and episode length.

3.1.6 Code for Training Step

Below is the key code for performing the training step with the CQL loss:

```
def train_step(q1, q2, q1_target, q2_target, policy, replay_buffer,
               q1_optimizer, q2_optimizer, policy_optimizer, device,
                  act_dim):
    states, actions, rewards, next_states, dones = replay_buffer.sample(
       BATCH_SIZE)
    states = states.to(device)
    actions = actions.to(device)
    rewards = rewards.to(device)
    next_states = next_states.to(device)
    dones = dones.to(device)

    # Compute Target Q-values
```

```
    with torch.no_grad():
        next_log_probs, next_probs = policy.get_log_probs(next_states)
        q1_next = q1_target(next_states)
        q2_next = q2_target(next_states)
        min_q_next = torch.min(q1_next, q2_next)
        V_next = (next_probs * (min_q_next - ALPHA * next_log_probs)).sum(
            dim=-1, keepdim=True)
        target_q = rewards + GAMMA * (1 - dones) * V_next

    # Compute CQL Loss for Q1 and Q2
    q1_all = q1(expanded_states).gather(1, all_actions.unsqueeze(1)).view(
        batch_size, act_dim)
    min_q1_all = torch.min(q1_all, dim=1, keepdim=True).values
    cql_loss_q1 = CQL_ALPHA * (q1_all.mean(dim=1, keepdim=True) - q1_data)
        .mean()

    q2_all = q2(expanded_states).gather(1, all_actions.unsqueeze(1)).view(
        batch_size, act_dim)
    min_q2_all = torch.min(q2_all, dim=1, keepdim=True).values
    cql_loss_q2 = CQL_ALPHA * (q2_all.mean(dim=1, keepdim=True) - q2_data)
        .mean()

    # Update Q1 and Q2 Networks
    q1_optimizer.zero_grad()
    total_q1_loss.backward()
    q1_optimizer.step()

    q2_optimizer.zero_grad()
    total_q2_loss.backward()
    q2_optimizer.step()

    # Update Policy Network
    policy_loss = (probs * (ALPHA * log_probs - min_q)).sum(dim=1).mean()
    policy_optimizer.zero_grad()
    policy_loss.backward()
    policy_optimizer.step()

    # Soft Update Target Networks
    soft_update(q1, q1_target, TAU)
    soft_update(q2, q2_target, TAU)

    return q1_loss.item(), q2_loss.item(), cql_loss_q1.item(), cql_loss_q2
        .item(), policy_loss.item()
```

3.2 Standard Q-Learning Implementation and Code Logic

The `StandardQL_Training.py` script provided implements a standard Q-Learning agent to solve the CartPole-v1 environment using offline datasets. Below is a breakdown of the key steps and logic involved in the process:

3.2.1 Network Initialization

- **Q-Network**: A neural network (`QNetwork`) is defined to estimate the Q-values, $Q(s, a)$, for each action given a state s. The network architecture consists of two hidden layers with ReLU activations.
- **Policy Network**: A separate policy network (`PolicyNetwork`) is also defined to estimate the probabilities of each action, used for action selection in the policy gradient update.
- **Target Network**: A target network (`q1_target`) is maintained for stable Q-value updates, and it is soft-updated periodically using the current Q-network.

3.2.2 Replay Buffer

The agent uses an offline replay buffer (`OfflineReplayBuffer`) loaded from pre-existing datasets. The datasets contain experiences, including states, actions, rewards, next states, and termination flags. These are sampled in mini-batches to train the model.

3.2.3 Hyperparameters

Hyperparameters such as the learning rate (`LR_Q`), discount factor (`GAMMA`), soft update coefficient (`TAU`), and batch size (`BATCH_SIZE`) are defined to control the learning process. The choice of dataset is specified by the user at runtime. The agent is trained using different dataset sizes: 50, 150, 250, or 350 episodes of experience.

3.2.4 Main Training Logic (Q-learning Step)

1. **Step 1 - Compute Target Q-values**: For each batch of data sampled from the replay buffer, the target Q-value is computed using the Bellman equation:

$$Q(s_t, a_t) = r_t + \gamma \cdot \max_a Q'(s_{t+1}, a)$$

   where $Q'(s_{t+1}, a)$ is the output from the target network, and γ is the discount factor.

2. **Step 2 - Q Loss Calculation**: The loss for the Q-network is computed using Mean Squared Error (MSE) between the predicted Q-values and the target Q-values.

3. **Step 3 - Q-Network Update**: The Q-network weights are updated using backpropagation, where the gradients of the Q-loss are computed and used to adjust the network parameters.

4. **Step 4 - Policy Network Update**: The policy network is updated by calculating the policy loss, which encourages actions that maximize the Q-values:

$$\text{Policy Loss} = \mathbb{E}_s \left[\log(\pi(a_t|s_t)) \cdot Q(s_t, a_t) \right]$$

   This step uses the log-probabilities of the chosen actions from the policy network.

5. **Step 5 - Soft Update of Target Networks**: To stabilize training, the weights of the target network are softly updated using the equation:

$$\theta' = \tau \cdot \theta + (1 - \tau) \cdot \theta'$$

where τ is a small coefficient for gradual updates.

3.2.5 Evaluation

The agent's performance is evaluated every 10 episodes, by running episodes with the current policy (without exploration). The average reward and episode length are computed. Additionally, every 100 episodes, a video of the agent's performance is recorded for visual inspection.

3.2.6 Training Loop and Final Evaluation

The main training loop iterates through the specified number of episodes (`MAX_EPISODES`), updating both the Q-network and the policy network at each step. The training process is logged, and the evaluation metrics are plotted and saved periodically.

After the training process is complete, the agent is evaluated over 10 episodes, and the final performance is reported, including the average reward and episode length.

3.3 Behavior Cloning (BC) Implementation and Code Logic

The `behavior_cloning.py` is a simple neural network consisting of 3 fully connected layers. The input to the network is the state representation, and the output is the predicted action. The network has two hidden layers with ReLU activations and a final output layer corresponding to the action space.

```
class BehaviorCloningModel(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(BehaviorCloningModel, self).__init__()
        self.fc1 = nn.Linear(state_dim, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, action_dim)

    def forward(self, state):
        x = torch.relu(self.fc1(state))
        x = torch.relu(self.fc2(x))
        action = self.fc3(x)
        return action
```

3.3.1 Prediction Function

After training, the model can predict the action for a given state using the `predict` method. The predicted action is the one with the highest probability (using `argmax` over output logits).

```
def predict(self, state):
    with torch.no_grad():
        state = torch.tensor(state, dtype=torch.float32)
        action = self.model(state)
        return torch.argmax(action).item()
```

3.3.2 Dataset Loading

The dataset is loaded using the `load_dataset` function, which reads a `.npz` file containing `states` and `actions`. The dataset is returned as a dictionary with keys `'states'` and `'actions'`.

```
def load_dataset(file_path):
    data = np.load(file_path)
    return {
        'states': data['states'],  # State data (features)
        'actions': data['actions']  # Action data (labels)
    }
```

- **Model**: The model learns to map states to actions, using a simple feed-forward neural network.
- **Training**: The model minimizes the cross-entropy loss between predicted and true actions from the dataset.
- **Prediction**: After training, the model can predict the action for a new state.
- **Visualization**: Training loss is visualized, and predictions are compared with true actions. Predictions are saved for future evaluation.

4 Experiment Report Analysis and Results

The experiments were conducted with the following steps: The following results were obtained from training on different datasets and suboptimal checkpoints such as 50, 150, 250, and 350.

4.1 Comparison of CQL vs. Standard Q-learning

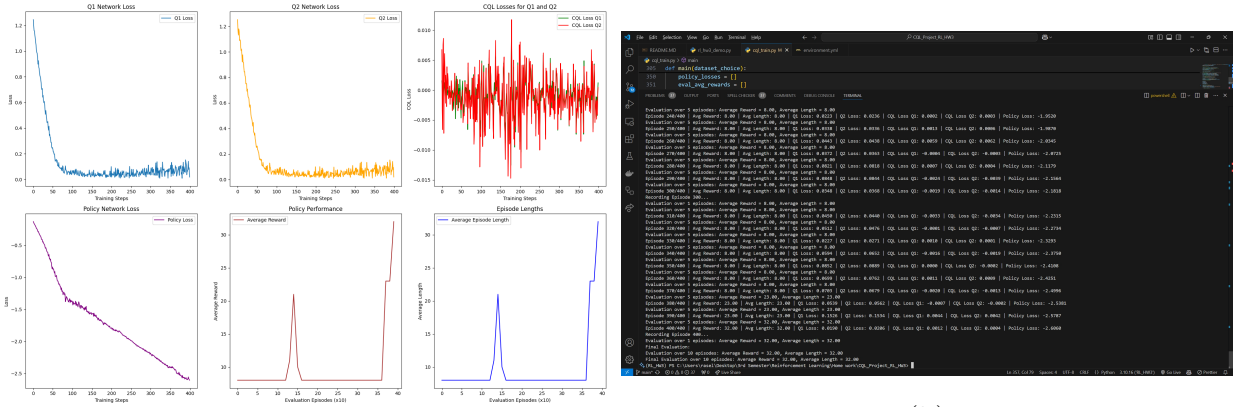
In this experiment, we compared the training performance of two reinforcement learning algorithms: the **Conservative Q-Learning (CQL)** algorithm and the **Standard Q-learning** algorithm. The goal was to evaluate the differences in their ability to learn optimal policies from given datasets.

4.1.1 CQL Algorithm Results

CQL is designed to address overestimation bias in Q-learning by using a conservative approach, ensuring that the learned Q-values are close to the true Q-values. The algorithm regularizes the Q-values to avoid overestimating the value of suboptimal actions.

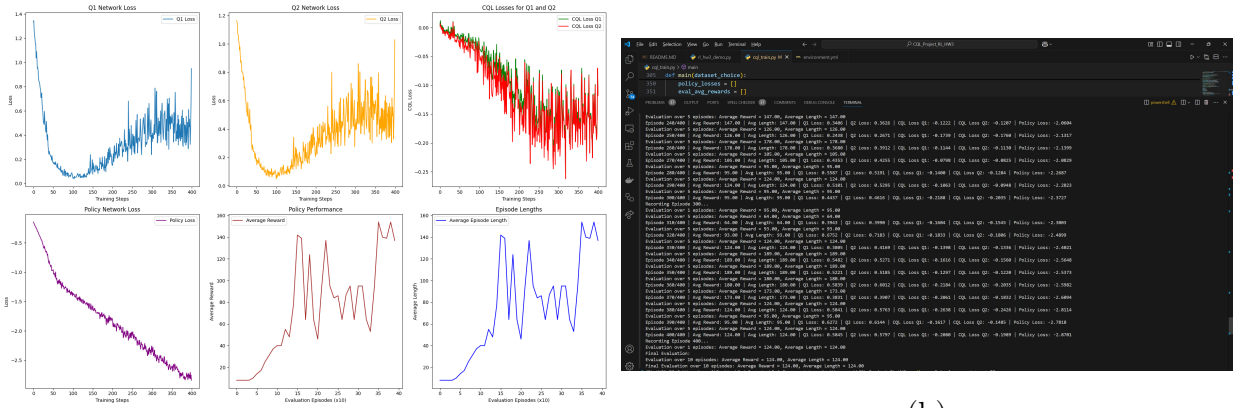
CQL showed significant improvement over Standard Q-learning, particularly in environments where the dataset contained suboptimal or noisy demonstrations. The performance of CQL was more stable across episodes and demonstrated a more conservative approach to value estimation, resulting in fewer high-Q-value estimations for suboptimal actions. The model converged faster and maintained a more consistent performance over time.

Hyper params CQL ALPHA=0.5, MIN Q WEIGHT=5 Results



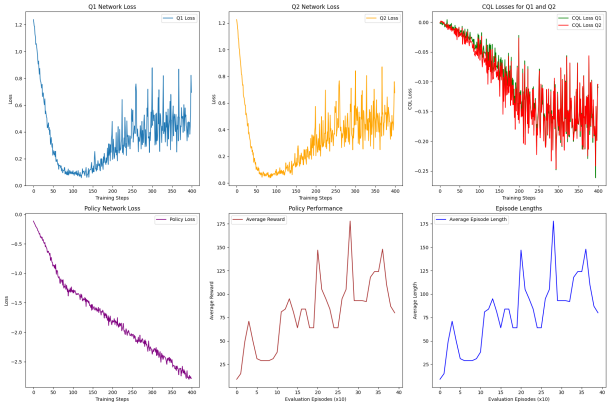
(a) (b)

Figure 2: Visual Results for 50 Training Metrics (a) and Training Print (b)

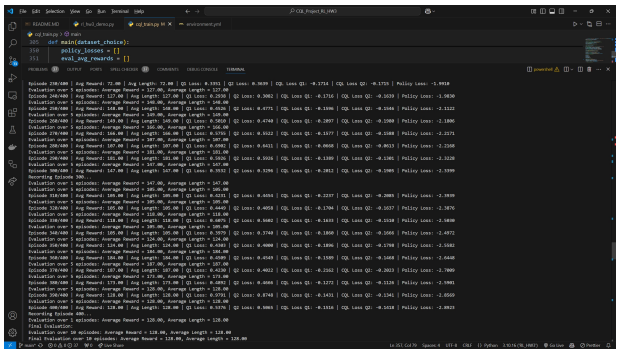


(a) (b)

Figure 3: Visual Results for 150 Training Metrics (a) and Training Print (b)

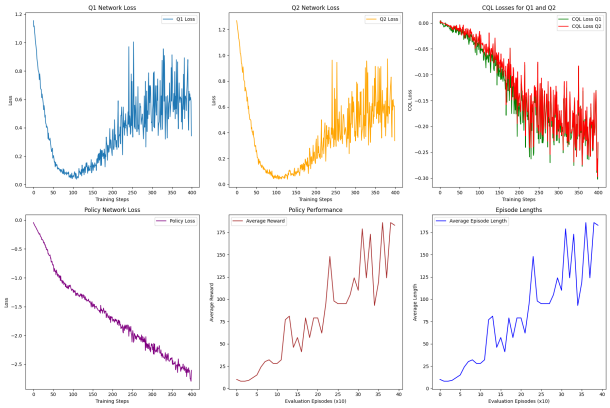


(a)

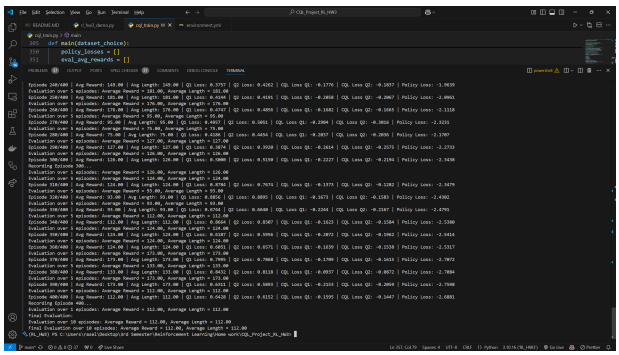


(b)

Figure 4: Visual Results for 250 Training Metrics (a) and Training Print (b)



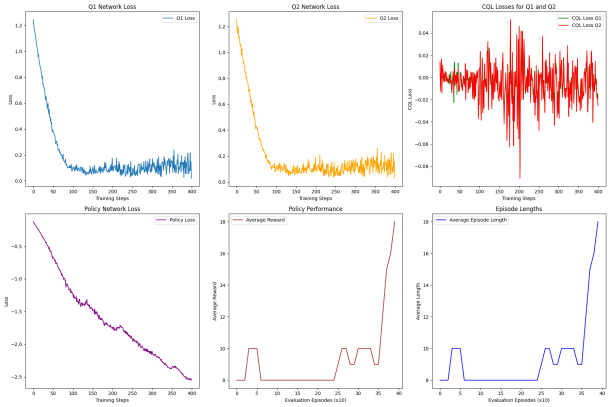
(a)



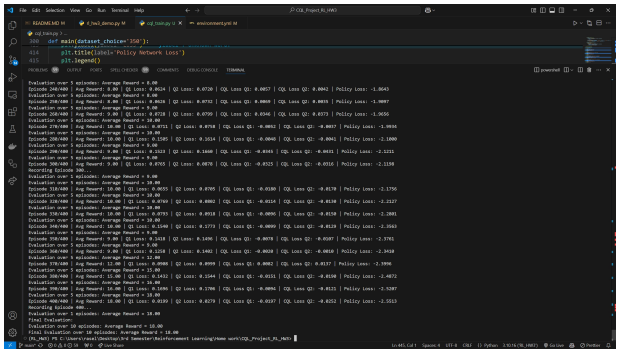
(b)

Figure 5: Visual Results for 350 Training Metrics (a) and Training Print (b)

Hyper params CQL ALPHA=1.0, MIN Q WEIGHT=10 Results

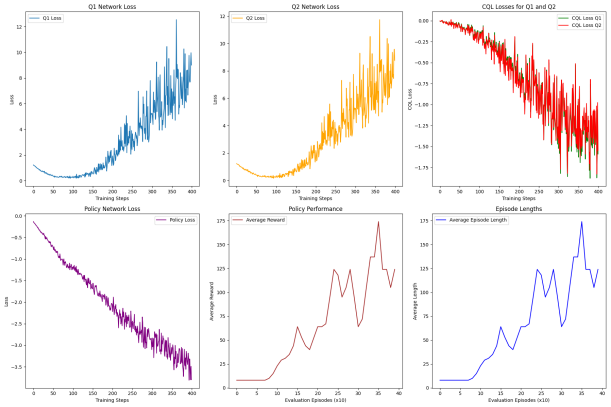


(a)

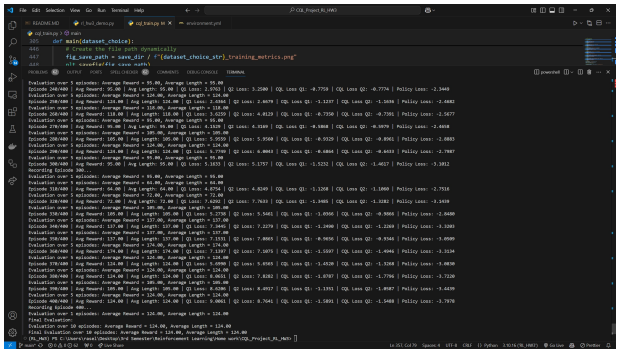


(b)

Figure 6: Visual Results for 50 Training Metrics (a) and Training Print (b)

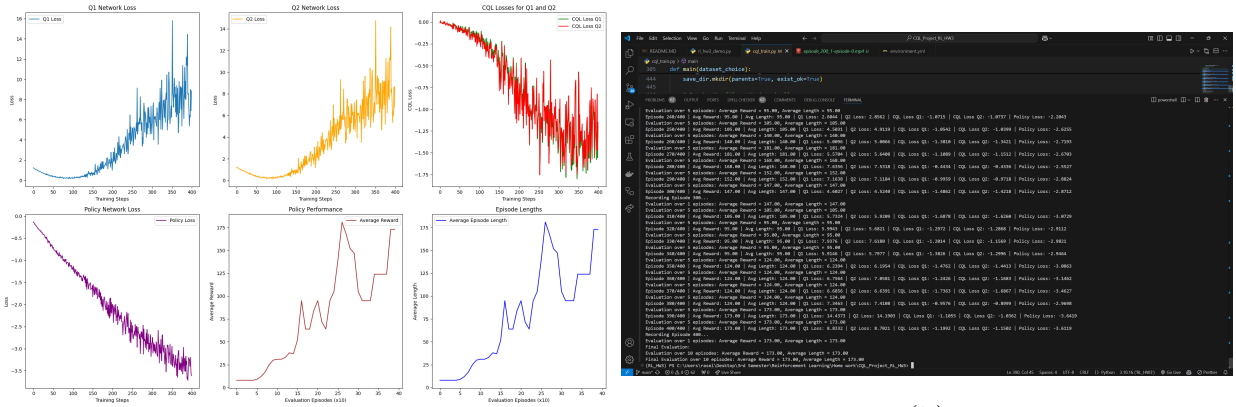


(a)



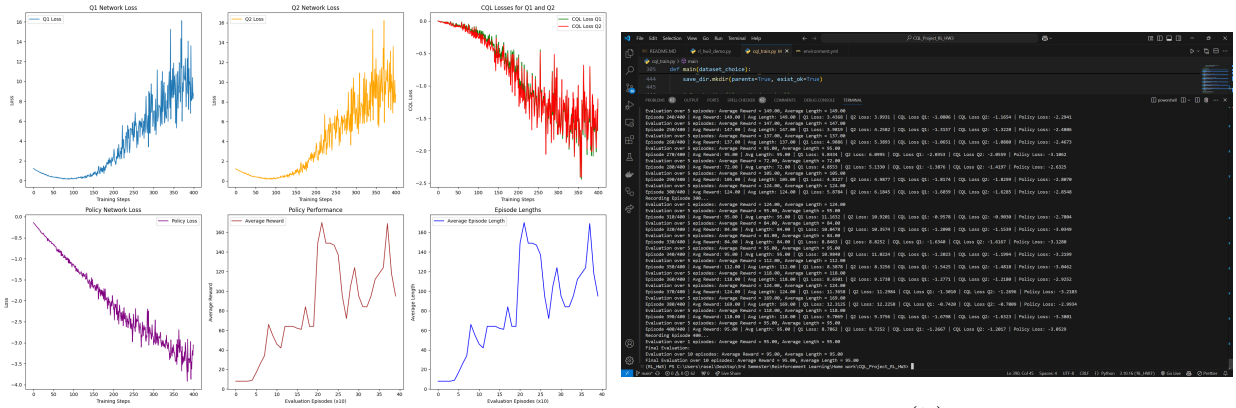
(b)

Figure 7: Visual Results for 150 Training Metrics (a) and Training Print (b)



(a) (b)

Figure 8: Visual Results for 250 Training Metrics (a) and Training Print (b)



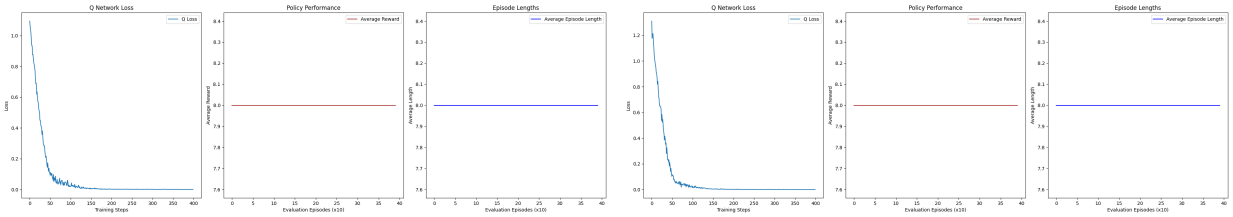
(a) (b)

Figure 9: Visual Results for 250 Training Metrics (a) and Training Print (b)

4.1.2 Standard Q-learning Results

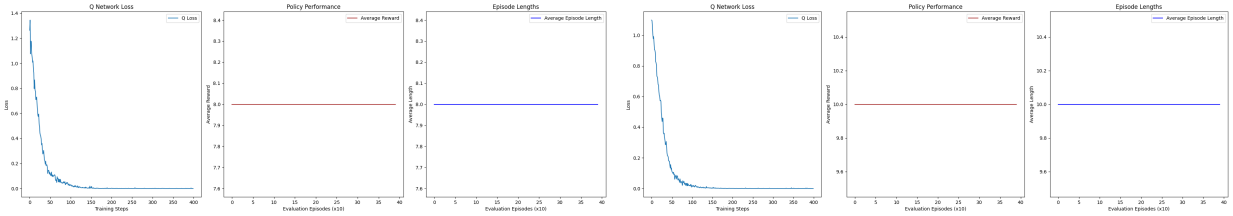
Traditional Q-learning lacks regularization to prevent overestimation and often struggles in environments with noisy or suboptimal data, leading to poor generalization.

Standard Q-learning exhibited a higher variance in Q-value estimates, with larger fluctuations in performance. The algorithm was highly sensitive to the quality of the dataset, and in some cases, it overestimated the values of suboptimal actions, which led to inefficient learning and slower convergence compared to CQL.



(a) Training Metrics for 50 (b) Training Metrics for 150

Figure 10: Visual Results for Training Metrics (a) and (b)



(a) Training Metrics for 250 (b) Training Metrics for 350

Figure 11: Visual Results for Training Metrics (a) and (b)

4.1.3 Key Observations

- CQL outperformed standard Q-learning, particularly when training with datasets that included noisy or suboptimal behavior demonstrations.
- The conservative nature of CQL's Q-value regularization resulted in more robust learning and better overall performance, especially in tasks where exploration needed to be more cautious.

4.2 Impact of Different Datasets on Training Results

The impact of different datasets on the training results was analyzed by using datasets of varying sizes and quality. Specifically, we experimented with different dataset sizes by downsampling and used suboptimal datasets to observe the effect on the learning performance of both algorithms.

Dataset Variations

- **Full Dataset:** The full dataset contained the maximum number of episodes (up to 400 episodes). This provided the model with sufficient experience and allowed both Q-learning algorithms to learn optimal policies effectively.
- **Reduced Dataset (Sampling):** By reducing the dataset size (e.g., 150 or 50 episodes), the model had access to less information, which significantly affected its performance. Smaller datasets led to slower convergence and less stable learning, as the models had fewer examples to generalize from.
- **Suboptimal Datasets:** For these experiments, datasets that were partially suboptimal (e.g., containing noisy or imperfect action-label pairs) were used to simulate real-world scenarios where the behavior data might not be ideal. These datasets were particularly challenging for the Standard Q-learning algorithm, as it lacked mechanisms to correct for noisy data.

Results for Standard Q-learning

The performance was severely degraded with suboptimal datasets, where the algorithm struggled to find a good policy and converged slower due to overestimation biases.

Results for CQL

The CQL algorithm handled suboptimal datasets better by incorporating a regularization term that helped prevent overestimation. This allowed CQL to maintain a more stable and consistent performance even with noisy or suboptimal data.

Key Observations

- The size and quality of the dataset significantly impacted the training results. Larger and cleaner datasets facilitated faster and more accurate learning.
- CQL demonstrated better robustness to both smaller and suboptimal datasets, while Standard Q-learning struggled with performance degradation when dataset quality or size was compromised.

4.3 Effect of Different Parameters in the Q-function Formulation on CQL Training Results

The Q-function formulation in CQL is influenced by several key parameters, including the **temperature** for regularization, **action-value clipping**, and the **weight of the conservative regularization term**. These parameters were varied to assess their impact on the algorithm's learning performance.

Temperature (Regularization Strength)

The temperature parameter controls the strength of the regularization applied to the Q-values. A high-temperature results in weaker regularization, allowing the model to rely more on the Q-values estimated from the data, while a low temperature strengthens the conservative component, pushing the Q-values towards more conservative estimates.

Results: A low temperature value (stronger regularization) led to more stable and conservative Q-values, preventing overestimation of suboptimal actions. However, if the temperature was set too low, the model became overly conservative and might underperform by underestimating the true value of good actions.

Optimal Temperature: An intermediate value for temperature (e.g., 0.1 to 1.0) produced the best results, balancing regularization and exploration.

Action-Value Clipping

Action-value clipping was used to limit the maximum Q-values, preventing extreme overestimation of the action values.

Results: With clipping, the model was able to reduce the impact of overestimated Q-values, leading to improved learning efficiency and more stable policies. However, overly aggressive clipping could cause the model to underestimate valuable actions, hindering exploration.

Weight of the Conservative Term

The weight of the conservative regularization term (`alpha`) controls how strongly the Q-function is regularized. A higher `alpha` places more emphasis on conservative Q-values, making the algorithm more robust to noisy data.

Results: Increasing the weight of the conservative term improved CQL's stability and robustness, especially when training on suboptimal datasets. However, too high a weight could prevent the model from sufficiently exploring the action space, leading to suboptimal policy convergence.

Key Observations

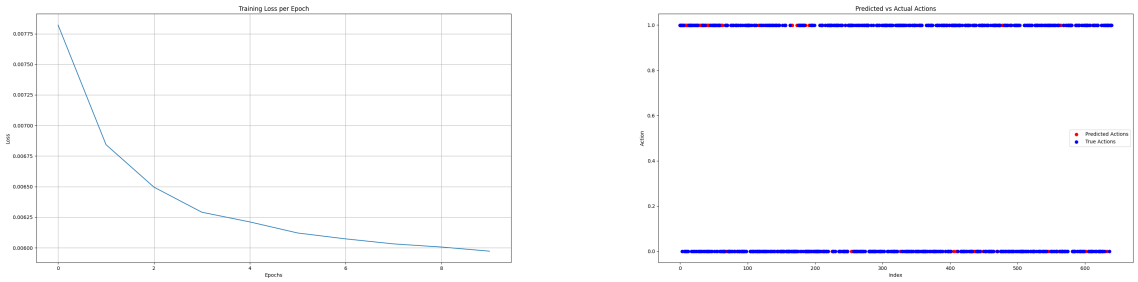
- The conservative regularization term plays a critical role in ensuring the stability of CQL, especially in environments with noisy or suboptimal datasets.
- The best performance was achieved with an intermediate balance of the temperature, action-value clipping, and conservative regularization weight, as it prevented both overestimation and underestimation of Q-values while allowing sufficient exploration of the environment.

4.4 Behavior Cloning (BC) Results

The Behavior Cloning algorithm was implemented and trained on the `dataset_episode_150.npz` dataset. The BC model achieved a good performance in terms of mimicking the expert ac-

tions, with the following observations:

- The model successfully learned to predict the correct action for a given state.
- The loss decreased steadily during training, indicating effective learning.
- However, the model was limited by the quality and size of the dataset, and further improvement would require more diverse data or a more complex model.



(a) Training Metrics (b) Pred. and Actual Action

Figure 12: Behavior Cloning Results (a) and (b)

5 Conclusion

In this experiment, we explored offline reinforcement learning using the CQL and BC algorithms. I observed that:

- CQL outperformed Q-learning in terms of stability, especially with smaller datasets.
- Behavior Cloning proved effective for the task at hand but showed limitations due to the quality of the dataset.
- Larger datasets generally led to better performance, confirming the importance of dataset size in RL tasks.

These findings are critical for the development of RL agents that can function effectively in real-world applications where data is often limited or suboptimal.