ECE433/COS435 Introduction to RL
Assignment 2: MPC, CEM; Imitation Learning
Spring 2025

Fill me in

Your name here.

Due February 19, 2025

Collaborators

Fill me in

Please fill in the names and NetIDs of your collaborators in this section.

Instructions

Writeups should be typeset in Latex and submitted as PDF. You can work with whatever tool you like for the code, but **please submit the asked-for snippet and answer in the solutions box as part of your writeup. We will only be grading your writeup.**

Grading. Questions 1-2 will collectively be worth total 50 points, and question 3-4 will also be worth 50 points, making the total score 100 points.

Question 1 (25 points): MPC with CEM for Cart-Pole Control

You are required to implement a Model Predictive Control (MPC) approach for the CartPole-v1 environment. Within each MPC step, use the Cross-Entropy Method (CEM) to search over a horizon H for the best sequence of actions. Execute only the first action from that sequence in the real environment, then re-plan at the next step. You need to implement them from scratch.

1. Write two functions from scratch, with the input arguments provided for each function in parentheses:
   - `cross_entropy_method(mean_probs, horizon, num_actions, sample_size, elite_frac, cem_iterations, evaluate_fn)`
   - `mpc_control(env, horizon, cem_iterations, sample_size, elite_frac, gamma)`
2. `cross_entropy_method(...)`
   - Maintains or updates a probability distribution over action sequences (length H). For simplicity here, we only fit the mean of action sequence distributions, i.e., `mean_probs`, which provides its initial values as input argument and should be returned as output after the whole update process. Please note that it should be a distribution over sequences of horizon H instead of distribution over actions at one step.
   - We consider a simple 1-dimensional discrete action space with `num_actions` actions. As an example, if `num_actions`=2, the two actions in the action space are integers: 0 and 1. If `num_actions`=4, the four actions in the action space are integers: 0, 1, 2 ,3.
   - Samples multiple candidates according to `samle_size`, evaluates them with `evaluate_fn` which yields one return for one sequence, selects "elite" sequences with high returns according to the `elite_frac` which is the ratio of "elite" sequences over the all sampled sequences, and refits the distribution over the "elite" sequences.
   - Iterates above steps to fit the distributions for `cem_iterations` steps. Returns the best sequence found over all update iterations and the final `mean_probs` (which describes the final distribution).
3. `mpc_control(env, horizon, cem_iterations, sample_size, elite_frac, gamma)`
   - On each environment step, calls the above CEM to get an action sequence of length H from the current state.
   - Executes the first action in the real environment, obtains next state, and continues until done.
   - The `env` is an environment instantiation with two functions provided:

     At the beginning, the environment is reset and yields an observation.

     ```
     obs , _ = env.reset()
     ```

     At each step, the environment is taking an action and transits to next state (with an observation and a reward yielded), and the "done" is a boolean value describing whether the environment simulation finishes:

     ```
     obs_next , rew , done , _, _ = env.step(action)
     ```

   - You will need to use subroutines `cross_entropy_method(...)` (as you write in last question) and `evaluate_fn`. The `evaluate_fn` is provided and you can use directly:

     ```
             def evaluate_fn(seq):
                 temp_env = gym.make('CartPole-v1', render_mode=None)
                 temp_env.reset()
                 total_ret = 0.0
                 discount_factor = 1.0
                 for action in seq:
                     # step
                     s_next , rew , done_ , _, _ = temp_env.step(action)
                     total_ret += discount_factor * rew
     ```

```python
                discount_factor *= gamma
                if done_:
                    break
            return total_ret
```

Notes: In your solution, carefully implement how you evaluate each candidate action sequence from the current state. The complete code solution can be found at colab.

Solution

```python
def cross_entropy_method(mean_probs, horizon, num_actions, sample_size, elite_frac,
    cem_iterations, evaluate_fn):
    """
    Cross-Entropy Method (CEM) to search for the best action sequence of length '
    horizon'.
    mean_probs shape: (horizon, num_actions).
    """
    best_seq = None
    best_return = -1e9

    n_elite = max(1, int(sample_size * elite_frac))

    # Repeatedly refine the distribution
    for _ in range(cem_iterations):
        # (a) Sample 'sample_size' sequences
        sequences = []
        for _s in range(sample_size):
            seq = []
            for t in range(horizon):
                # Draw action from the distribution at time t
                act = np.random.choice(num_actions, p=mean_probs[t])
                seq.append(act)
            sequences.append(seq)
        sequences = np.array(sequences)  # shape: (sample_size, horizon)

        # (b) Evaluate each sequence
        returns = []
        for seq in sequences:
            ret = evaluate_fn(seq)
            returns.append(ret)
        returns = np.array(returns)

        # (c) Select top 'elite_frac'
        elite_inds = np.argsort(returns)[-n_elite:]  # indices of best returns
        elite_seqs = sequences[elite_inds]
        elite_returns = returns[elite_inds]

        # Update the best overall
        elite_best_return = elite_returns.max()
        if elite_best_return > best_return:
            idx = np.argmax(elite_returns)
            best_return = elite_best_return
            best_seq = elite_seqs[idx]

        # (d) Refit distribution by counting action frequencies among elites
        new_probs = np.zeros_like(mean_probs)
        for t in range(horizon):
            actions_t = elite_seqs[:, t]  # shape (n_elite,)
            for a in range(num_actions):
                new_probs[t, a] = np.sum(actions_t == a)
            # Normalize
            new_probs[t] /= (new_probs[t].sum() + 1e-8)

        mean_probs = new_probs

    # Return the best sequence found
    return best_seq, mean_probs


def mpc_control(env, horizon=15, cem_iterations=4, sample_size=200, elite_frac=0.1,
    gamma=1.0):
    """
    Model Predictive Control (MPC) with CEM for the CartPole environment.
    """
    obs, _ = env.reset()
    done = False
    total_reward = 0.0
    step_count = 0
    max_steps = env._max_episode_steps
    num_actions = env.action_space.n  # should be 2 for CartPole

    # Initialize uniform distribution: shape (horizon, num_actions)
    mean_probs = np.ones((horizon, num_actions)) / num_actions

    while not done and step_count < max_steps:

        # Evaluate function: returns how good a candidate sequence is from the
    current obs
        def evaluate_fn(seq):
            temp_env = gym.make('CartPole-v1', render_mode=None)
            temp_env.reset()
            total_ret = 0.0
            discount_factor = 1.0
            for action in seq:
                # step
                s_next, rew, done_, _, _ = temp_env.step(action)
                total_ret += discount_factor * rew
                discount_factor *= gamma
                if done_:
                    break
            return total_ret

        # (1) Use cross_entropy_method to find the best action sequence
        best_seq, mean_probs = cross_entropy_method(
            mean_probs,
            horizon,
            num_actions,
            sample_size,
            elite_frac,
            cem_iterations,
            evaluate_fn
        )

        # (2) Execute the first action
        action = best_seq[0]
        obs_next, rew, done, _, _ = env.step(int(action))
        total_reward += rew

        # (3) Move time forward
        obs = obs_next
        step_count += 1

    return total_reward
```

Question 2 (25 points): Zero-Order Policy Search using CEM

Now that you've seen how to use CEM for searching action sequences in a MPC example, let's re-use the same concept to search directly for a policy parameter vector. This is often called **"Zero-Order Policy Search,"** because we do not require gradients of the policy but use a zero-order optimization technique. We

only need to evaluate how good each parameter vector is.
In this question, you need to:

- Define a parametric policy for CartPole, which is an environment with a 4-dimensional state space $\mathbb{R}^4$.
- Reuse your CEM approach to search over $w \in \mathbb{R}^4$ as policy parameters.
- Evaluate each sampled w by running a full episode, gathering total reward, picking elites, and updating the distribution.
- After some iterations, obtain a final w. Evaluate that policy and report the average return over multiple episodes.

Below is a skeleton with TODOs for you to fill in.

Solution

```python
def choose_action(w, obs):
    """
    A simple linear policy for CartPole:
      action = 1 if w dot obs > 0 else 0.
    w, obs: shape (4,)
    """
    return 1 if np.dot(w, obs) > 0 else 0


def rollout_episode(env, w):
    """
    Run one full episode in 'env' using param 'w'.
    Return total (undiscounted) reward.
    """
    obs, _ = env.reset()
    done = False
    total_rew = 0
    while not done:
        action = choose_action(w, obs)
        obs, rew, done, _, _ = env.step(action)
        total_rew += rew
    return total_rew


def train_cem_policy(env, dim=4, cem_iterations=10, sample_size=50, elite_frac=0.2):
    """
    Zero-Order Policy Search with CEM over parameter vectors w in R^dim.
      - Initialize (mean, std).
      - For each iteration:
          1) Sample 'sample_size' param vectors
          2) Roll out each once, collect returns
          3) Select top 'elite_frac' fraction => new (mean, std)
      - Return best or final param.

    We'll add print statements to show training progress each iteration.
    """
    mean = np.zeros(dim)   # e.g. [0,0,0,0]
    std = np.ones(dim)*2.0 # can tune
    best_w = mean.copy()
    best_score = -1e9

    n_elite = max(1, int(sample_size * elite_frac))

    for iteration in range(cem_iterations):
        # 1) Sample
        params = np.random.randn(sample_size, dim)*std + mean

        # 2) Evaluate
        returns = []
        for i in range(sample_size):
            r = rollout_episode(env, params[i])
            returns.append(r)
        returns = np.array(returns)

        iteration_best = returns.max()
        if iteration_best > best_score:
            idx = np.argmax(returns)
            best_score = iteration_best
            best_w = params[idx].copy()

        # Print progress
        print(f"[CEM Policy Iter {iteration+1}/{cem_iterations}] best_in_iter={
    iteration_best:.2f}, overall_best={best_score:.2f}")

        # 3) Select elites
        elite_indices = np.argsort(returns)[-n_elite:]
        elite_params = params[elite_indices]  # shape (n_elite, dim)

        # 4) Update distribution
        mean = np.mean(elite_params, axis=0)
        std = np.std(elite_params, axis=0) + 1e-8

    return best_w


def eval_policy(env, w, episodes=5):
    """
    Evaluate a given policy param 'w' over multiple episodes,
    and return the average total reward.
    """
    total = 0
    for _ in range(episodes):
        ret = rollout_episode(env, w)
        total += ret
    return total / episodes
```

Question 3. (15 points). Flap Like How Flappy Sr. Taught You!

This is an introduction to implement Behavioral Cloning. The complete code and starter code can be found at colab.

(a) Policy Evaluation

Paste the **entire cell** implementing policy evaluation below.

Solution

```python
def evaluate_policy(env, policy, discount) -> float:
  """Returns a single-sample estimate of a policy's return.

  Args:
    env: OpenAI gym environment following old API.
    policy: Callable representing a policy.

  Returns:
    Single-sample estimate of return.
  """
  policy_return, cur_ob, done, n_steps = 0, env.reset(), False, 0
  while not done:
    action_dist = policy(cur_ob)
    action = action_dist.sample()
    cur_ob, reward, done, _ = env.step(action)
    policy_return += (discount ** n_steps) * reward
    n_steps += 1
  return policy_return

```

Paste the **entire cell** for evaluating a policy that chooses actions uniformly at random below.

Solution

```
def uniform_policy(_ob):
  return torch.distributions.Categorical(logits=torch.ones(env.action_space.n))

returns = [evaluate_policy(env, uniform_policy, 0.999) for _ in range(50)]
print(np.mean(returns))

```

(b) Defining a Policy

Paste the **entire cell** defining a policy class over discrete actions below.

Solution

```
class DiscretePolicy(nn.Module):
  def __init__(self, input_dim, n_actions):
    """Initializes a policy over the action set {0, 1, ..., n_actions-1}.

    Args:
      input_dim: Observation dimensionality.
      n_actions: Number of actions in environment.
    """
    super().__init__()
    self.linear1 = nn.Linear(input_dim, 64)
    self.linear2 = nn.Linear(64, 64)
    self.linear3 = nn.Linear(64, n_actions)

  def forward(self, ob) -> Distribution:
    """Returns a distribution over this policy's action set.
    """
    if isinstance(ob, np.ndarray):
      ob = torch.Tensor(ob)
    x = nn.functional.relu(self.linear1(ob))
    x = nn.functional.relu(self.linear2(x))
    logits = self.linear3(x)
    return torch.distributions.Categorical(logits=logits)

```

(c) Setting Up Behavioral Cloning

Paste the **entire cell** defining a behavioral cloning training loop below.

Solution

```
def train_policy_by_bc(policy, dataset, n_steps, batch_size) -> DiscretePolicy:
  """Trains the provided policy by behavioral cloning, by taking n_steps training
    steps with the
  provided optimizer. During training, training batches of size batch_size are
    sampled from the dataset
  to compute the loss.

  Args:
    policy: policy of class DiscretePolicy.
    dataset: The dataset, represented as a dictionary containing keys "obs"
      and "actions", mapping to arrays with observations and corresponding actions to
      clone, respectively. Arrays are of shape [dataset_size, ob_dim] and [
    dataset_size].
    n_steps: Number of training steps to take.
    batch_size: Size of the sampled batch for each training step.

  Returns:
    A policy trained according to the parameters above.
  """
  dataset_size = len(dataset["obs"])
  optimizer = torch.optim.Adam(policy.parameters(), lr=0.001)
  loss = nn.CrossEntropyLoss()
  for _ in range(n_steps):
    # Sample batch
    batch_idxs = np.random.randint(dataset_size, size=(batch_size))
    batch_ob = torch.Tensor(dataset["obs"][batch_idxs])
    batch_ac = torch.Tensor(dataset["actions"][batch_idxs]).type(torch.LongTensor)

    # Zero out optimizer gradients
    optimizer.zero_grad()

    # Get predictions
    action_dists = policy(batch_ob)

    # Compute loss and backpropagate gradients
    mean_loss = -torch.mean(action_dists.log_prob(batch_ac))
    mean_loss.backward()

    # Alternative way to compute the loss
    # mean_loss = loss(actions_dists.logits, batch_ac)

    # Adjust weights
    optimizer.step()

  return policy

```

(d) Setting Up Behavioral Cloning

Paste the **entire cell** applying behavioral cloning to `flappy_sr_notes.mat` below.

Solution

```
from scipy.io import loadmat

# Load dataset
dataset = loadmat("/content/flappy_sr_notes.mat")
dataset = {
    "obs": dataset["observations"],
    "actions": dataset["actions"][:, 0]
}

# Train policy by BC
policy = DiscretePolicy(env.observation_space.shape[0], env.action_space.n)
```

```python
policy = train_policy_by_bc(policy, dataset, 20000, 64)

# Evaluate
returns = [evaluate_policy(env, policy, 0.999) for _ in range(50)]
print(np.mean(returns))

```

Question 4 (15 points): Floppy the Sloppy Ruins (?) the Day (AKA An Introduction to Filtered Behavioral Cloning)

(a) What is Left???

Paste the **entire cell** applying behavioral cloning to `vandalized_notes.mat` below.

Solution

```python
from scipy.io import loadmat

dataset = loadmat("/content/vandalized_notes.mat")
dataset = {
    "obs": dataset["observations"],
    "actions": dataset["actions"][:, 0]
}

# Train policy by BC
policy = DiscretePolicy(env.observation_space.shape[0], env.action_space.n)
policy = train_policy_by_bc(policy, dataset, 20000, 64)

returns = [evaluate_policy(env, policy, 0.999) for _ in range(50)]
print(np.mean(returns))

```

(b) Array of Hope???

Paste the **entire cell** defining a reweighed behavioral cloning loss below.

Solution

```python
class ReweighedBCLossI(nn.Module):
  def __init__(self):
    super().__init__()
    self.base_loss = nn.CrossEntropyLoss(reduction='none')   # reduction='none' is
    very important here!

  def forward(self, batch_predictions, batch_targets, batch_weights):
    return torch.dot(self.base_loss(batch_predictions, batch_targets), batch_weights)

# Alternative implementation that passes in the action distribution directly
class ReweighedBCLossII(nn.Module):
  def __init__(self):
    super().__init__()

  def forward(self, batch_predictions, batch_targets, batch_weights):
    return -torch.dot(batch_predictions.log_prob(batch_targets), batch_weights)


```

Paste the **entire cell** defining a training loop using the reweighed behavioral cloning loss below.

Solution

```python

def train_policy_by_filtered_bc(policy, dataset, weights, n_steps, batch_size) -> DiscretePolicy:
  """Trains the provided policy by filtered behavioral cloning, by taking n_steps training steps with the
  provided optimizer with the provided weights. During training, training batches of size batch_size are sampled from the dataset
  to compute the loss.

  Args:
    policy: policy of class DiscretePolicy.
    dataset: The dataset, represented as a dictionary containing keys "obs"
      and "actions", mapping to arrays with observations and corresponding actions to
      clone, respectively. Arrays are of shape [dataset_size, ob_dim] and [dataset_size].
    weights: Weights used in the filtered BC loss.
    optimizer: An instance of torch.optim.Optimizer.
    n_steps: Number of training steps to take.
    batch_size: Size of the sampled batch for each training step.

  Returns:
    A policy trained according to the parameters above.
  """
  dataset_size = len(dataset["obs"])
  bc_loss1 = ReweighedBCLossI()   # Having both here to demonstrate how to use both implementations above
  bc_loss2 = ReweighedBCLossII()
  optimizer = torch.optim.Adam(policy.parameters(), lr=0.001)
  for _ in range(n_steps):
    # Sample dataset
    batch_idxs = np.random.randint(dataset_size, size=(batch_size,))
    batch_ob = torch.Tensor(dataset["obs"][batch_idxs])
    batch_ac = torch.Tensor(dataset["actions"][batch_idxs]).type(torch.LongTensor)
    batch_weights = torch.Tensor(weights[batch_idxs])

    # Zero out optimizer gradients
    optimizer.zero_grad()

    # Get predictions
    action_dists = policy(batch_ob)

    # Compute loss and backpropagate gradients
    mean_loss1 = bc_loss1(action_dists.logits, batch_ac, batch_weights)    # Implementation I
    mean_loss2 = bc_loss2(action_dists, batch_ac, batch_weights)           # Implementation II
    mean_loss2.backward()                                                  # Gonna use second here, but both should work.

    # Adjust weights
    optimizer.step()

  return policy

```

(c) Filtering Strategy I: Trajectory-Level Reweighting???

Paste the **entire cell** implementing the trajectory-level reweighting scheme below.

Solution

```python
from scipy.special import softmax

def trajectory_level_return_weights(dataset, temp, discount) -> np.ndarray:
  """Computes an array of weights for each point in the provided dataset according to
     the trajectory-level reweighting scheme.

  Args:
    dataset: Input dataset.
    temp: Temperature used in softmax.
    discount: Discount used to compute return.

  Returns:
    An array of weights for each BC datapoint in the dataset.
  """
  # Useful constants
  n_trajs = np.count_nonzero(dataset["is_rollout_start"])
  n_pts = len(dataset["is_rollout_start"])

  # Compute returns for each trajectory
  traj_returns = [0 for _ in range(n_trajs)]
  traj_idx = -1
  cur_traj_len = 0
  for idx in range(n_pts):
    if dataset["is_rollout_start"][idx]:
      traj_idx += 1
      cur_traj_len = 0
    traj_returns[traj_idx] += (discount ** cur_traj_len) * dataset["rewards"][idx]
    cur_traj_len += 1

  # Compute softmax weights
  traj_weights = softmax(np.array(traj_returns) / temp)

  # Assign to each datapoint
  traj_idx = -1
  final_weights = [0 for _ in range(n_pts)]
  for idx in range(n_pts):
    if dataset["is_rollout_start"][idx]:
      traj_idx += 1
    final_weights[idx] = traj_weights[traj_idx]

  return np.array(final_weights)

```

Paste the **entire cell** applying the reweighting scheme above to Filtered BC.

Solution

```python
dataset = loadmat("/content/vandalized_notes.mat")
dataset = {
    "obs": dataset["observations"],
    "actions": dataset["actions"][:, 0],
    "rewards": dataset["rewards"][:, 0],
    "is_rollout_start": dataset["is_rollout_start"][:, 0]
}
weights = trajectory_level_return_weights(dataset, 3.0, 0.999)

# Train policy by Filtered BC
policy = DiscretePolicy(env.observation_space.shape[0], env.action_space.n)
policy = train_policy_by_filtered_bc(policy, dataset, weights, 20000, 64)

returns = [evaluate_policy(env, policy, 0.999) for _ in range(50)]
print(np.mean(returns))
```

```
```

(d) Filtering Strategy II: Truncated Future Return???

Paste the **entire cell** implementing the truncated future return reweighting scheme below.

Solution

```python
from scipy.special import softmax

def truncated_future_return_weights(dataset, truncation_horizon, temp, discount) ->
    np.ndarray:
  """Computes an array of weights for each point in the provided dataset according to
     the truncated future return reweighting scheme.

  Args:
    dataset: Input dataset.
    truncation_horizon: How many timesteps to consider for computing future return.
    temp: Temperature used in softmax.
    discount: Discount used to compute return.

  Returns:
    An array of weights for each BC datapoint in the dataset.
  """
  n_pts = len(dataset["is_rollout_start"])

  # Compute truncated returns from each datapoint
  # Note: Not the most efficient implementation - can use numpy's sliding window
    function with some
  #   clever transformation of a sliding window view of is_rollout_start array to
    vectorize this.
  all_returns = [0 for _ in range(n_pts)]
  for idx in range(n_pts):
    for offset in range(truncation_horizon):
      if offset > 0 and (idx + offset >= n_pts or dataset["is_rollout_start"][idx +
    offset]):
        break
      all_returns[idx] += (discount ** offset) * dataset["rewards"][idx + offset]

  softmax_weights = softmax(np.array(all_returns) / temp)
  return n_pts * softmax_weights

```

Paste the **entire cell** applying the reweighting scheme above to Filtered BC.

Solution

```python
from scipy.io import loadmat

dataset = loadmat("/content/vandalized_notes.mat")
dataset = {
    "obs": dataset["observations"],
    "actions": dataset["actions"][:, 0],
    "rewards": dataset["rewards"][:, 0],
    "is_rollout_start": dataset["is_rollout_start"][:, 0]
}
weights = truncated_future_return_weights(dataset, 100, 3.0, 0.999)

# Train policy by Filtered BC
policy = DiscretePolicy(env.observation_space.shape[0], env.action_space.n)
policy = train_policy_by_filtered_bc(policy, dataset, weights, 20000, 64)

returns = [evaluate_policy(env, policy, 0.999) for _ in range(50)]
print(np.mean(returns))

```

Question 5 (20 points): Short-Answer Questions

For this problem, we will ask you a few questions regarding the experiments you ran to hopefully further develop your intuition for BC/Filtered BC. Limit your answer to each part to 2–3 sentences. You only need to choose 5 questions to answers. Note that these questions are found throughout Problem 2 in the provided `.ipynb` file.

1. What does the result of the experiment in Problem 2(a) tell you about running behavioral cloning on noisy/low-quality datasets? Intuitively, why does this happen?
(Hint: Think about what the BC loss is optimizing.)

Solution

The behavior cloning loss, effectively a negative total log-likelihood of the expert actions, is minimized when the policy faithfully predicts the same actions as the expert. If we run behavioral cloning on a noisy or low-quality expert dataset, it makes sense that the algorithm learns to replicate those bad state-to-action mappings, hurting the performance when we later evaluate the algorithm.

2. Why does the trajectory-level reweighting scheme in Problem 2(c) work? How does it affect what the BC loss is doing?

Solution

The trajectory-level reweighting scheme helps us determine which trajectories ended up with high returns and which didnt, and the softmax allows us to give less weight to low-reward trajectories and more weight to high-reward trajectories (which are likely to be high quality, low-noise expert data). In this sense, we can weaken the influence of noise on the behavioral cloning dataset, by decreasing the strength of the negative log-likelihood BC loss for datapoints that come from noisy trajectories. This approach effectively filters out datapoints that came from low-quality trajectories (as they only weakly contribute to the total loss), which allows filtered behavioral cloning to ignore Floppys low-quality data and focus on Flappy Srs high quality data.

3. What is the effect of the temperature on the weighing scheme in Problem 2(c)?
(Hint: As a starting point, think about what happens to the softmax function as $\alpha \to 0$. To make it even easier to think about, consider applying the softmax to two fixed values a, b with $a > b$ as you take this limit.)

Solution

If $\alpha \to 0$, only datapoints from the best (highest reward) trajectory will be assigned weight 1, and all others will get weight 0, which is underfitting because we dont train on all the high-quality data we have available. At the other extreme, $\alpha \to \infty$, every trajectory gets equal weight (equivalent to unfiltered BC), defeating the purpose of weighting the trajectories. As we increase the temperature, we allow trajectories other than the best one to increase their weights, which is a balancing act, as we want to up-weight high-quality trajectories while down-weighting low-quality ones.

4. One could consider a version of the reweighting scheme from Problem 2(c), where we remove the softmax and simply define the weight for τ_i as $R(\tau_i)$. While this may work in certain circumstances, can you think

of some potential pitfalls of such a weighing scheme?
(Hint: Think about the values the reward function could take in all kinds of environments).

Solution

Removing the softmax could result in a few pitfalls:
It could result in negative rewards and thus negative weights. Negative weights in a learn- ing algorithm can cause unexpected behavior, potentially inverting the importance of certain demonstrations.
The reward function for different trajectories can have very different scales. Without the normalization that softmax provides, the learning process may become sensitive to rewards with a very high value and lead to numerical instability.
Softmax ensure normalization by making all the weights sum to be 1 which wont be the case if we get rid of it.
Trajectories with zero rewards would be ignored and get assigned a weight of zero.
We wont have access to the tunable temperature perimeter anymore which can affect the trajectory rewards significantly.

5. Let us explore the effect of the hyperparameter T on the weighing scheme in Problem 2(d). Give a succinct description of the weights when $T = 1$. Do you expect this to work well in general? Why or why not?

Solution

When $T = 1$, the truncated future returns are just the rewards at those timesteps - this makes the return estimates myopic, and as a result, datapoints with high reward will be up-weighted while lower reward datapoints will be down-weighted. I dont expect this to work for general problems, as it ignores the long-term consequences of actions. For example, in chess, taking opponent pieces may have high short-term rewards, but it may result in a worse position for the endgame - behavioral cloning would only focus on high-reward actions, without considering strategic expert actions with lower immediate rewards but higher long-term payoffs

6. Can you think of a reason why one can set T in Problem 2(d) relatively small in Flappy Bird and still obtain decent performance?

Solution

A game like Flappy Bird primarily relies on short-term decision making. Each action (flap or not flap) has an immediate consequence, and the state of the game changes rapidly. We have immediate feedback available for each action in the environment. Rewards are also given promptly and frequently. This means that even with a small T, the agent can capture meaningful reward information. The action space is also limited. There are only two possible actions at any given time - flap or dont flap. This simplifies the decision-making process and reduces the need for long-term planning.

7. What are the risks of making an error when reconstructing Flappy Sr.s flight path? How does this compare to real-world applications like self-driving cars?

Solution

In many situations of the game going up or going down slightly early or later might not matter too much as the flappy bird can still overcome the obstacle. This is clearly not the case for self-driving cars where turning in one direction too early can have fatal consequences. In addition, every flappy bird episode is very similar so even if there is a mistake in the reconstruction of a single flight path, it can be considered as noise that will get averaged out. In the case of self-driving cars the situation is much more highly dimensional (speed, light, rain, road, bystanders, other cars, surroundings, inclination of road, car condition, etc. etc.) and therefore we might have a fairly small number of datapoints for each state which makes it more difficult to average out faulty behavior if that state appears in the deployment stage.

8. How can filtering strategies prevent a self-driving car from learning dangerous driving habits from human demonstrations?

Solution

In the process of creating data for self-driving cars, human drivers are likely to commit certain rule infringements like speeding or forgetting to blink, etc (not out of malevolence but simply because no human is perfect.) Post-hoc filtering strategies can help by either eliminating undesirable state-action pairs (like speeding) or assigning a low/negative reward to them herewith trying to reduce the probability of self-driving cars selecting that action

9. What ethical considerations arise in behavioral cloning when using data from competitors?

Solution

There are too many ethical issues that arise here that could be discussed. Here is a list to name a few: 1. Where does copying/violating intellectual property start and where does taking inspiration end? 2. If I train a model using a competitors data, does copyright/intellectual property lie in the process of training (aka developing an architecture, using certain code, etc.) or obtaining the actual underlying data used for training or both? 3. When company A queries company Bs model and then applies BC to the answers that the model provided, who owns the answers produced by the model? It could be company A, or B or even the owners of the data that was originally used by company B to train the model. 4. Lets say, in a imaginary world, a company called ClosedAI violated intellectual property rights to access data to train their model. If a second company uses that model to generate data for BC, is that second company liable for the same crime as ClosedAI? In other words, is stealing for thieves okay? 5. For the politically inclined: What is the value we give intellectual property (creators of the data). Private and intellectual property are very ingrained in western cultures but can (and maybe should?) be questioned.