```
%pip uninstall -y torch torchvision torchaudio
%pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cpu
```

```
WARNING: Skipping torch as it is not installed.
WARNING: Skipping torchvision as it is not installed.
WARNING: Skipping torchaudio as it is not installed.
Looking in indexes: https://download.pytorch.org/whl/cpu
Collecting torch
  Downloading https://download-r2.pytorch.org/whl/cpu/torch-2.11.0%2Bcpu-cp312-cp312-manylinux_2_28_x86_64.whl.metadata (29
Collecting torchvision
  Downloading https://download-r2.pytorch.org/whl/cpu/torchvision-0.26.0%2Bcpu-cp312-cp312-manylinux_2_28_x86_64.whl.metadat
Collecting torchaudio
  Downloading https://download-r2.pytorch.org/whl/cpu/torchaudio-2.11.0%2Bcpu-cp312-cp312-manylinux_2_28_x86_64.whl.metadata
Requirement already satisfied: filelock in /usr/local/lib/python3.12/dist-packages (from torch) (3.25.2)
Requirement already satisfied: typing-extensions>=4.10.0 in /usr/local/lib/python3.12/dist-packages (from torch) (4.15.0)
Requirement already satisfied: setuptools<82 in /usr/local/lib/python3.12/dist-packages (from torch) (75.2.0)
Requirement already satisfied: sympy>=1.13.3 in /usr/local/lib/python3.12/dist-packages (from torch) (1.14.0)
Requirement already satisfied: networkx>=2.5.1 in /usr/local/lib/python3.12/dist-packages (from torch) (3.6.1)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.12/dist-packages (from torch) (3.1.6)
Requirement already satisfied: fsspec>=0.8.5 in /usr/local/lib/python3.12/dist-packages (from torch) (2025.3.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages (from torchvision) (2.0.2)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /usr/local/lib/python3.12/dist-packages (from torchvision) (11.3.0)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.12/dist-packages (from sympy>=1.13.3->torch) (1.
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.12/dist-packages (from jinja2->torch) (3.0.3)
Downloading https://download-r2.pytorch.org/whl/cpu/torch-2.11.0%2Bcpu-cp312-cp312-manylinux_2_28_x86_64.whl (190.3 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 190.3/190.3 MB 3.9 MB/s eta 0:00:00
Downloading https://download-r2.pytorch.org/whl/cpu/torchvision-0.26.0%2Bcpu-cp312-cp312-manylinux_2_28_x86_64.whl (1.9 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.9/1.9 MB 44.5 MB/s eta 0:00:00
Downloading https://download-r2.pytorch.org/whl/cpu/torchaudio-2.11.0%2Bcpu-cp312-cp312-manylinux_2_28_x86_64.whl (341 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 341.3/341.3 kB 17.1 MB/s eta 0:00:00
Installing collected packages: torchaudio, torch, torchvision
Successfully installed torch-2.11.0+cpu torchaudio-2.11.0+cpu torchvision-0.26.0+cpu
```

CX Dialogue RL

I start by recreating the escalation issue with a small synthetic env. Goal: resolve conversations with less escalation and good speed. I run a baseline, read the logs, then iterate.

Quick paper read

Before coding, I skim two RL dialogue papers to keep reward shaping grounded. They show that reward choice and exploration drive behavior.

- Li et al. 2016: long term reward changes dialogue behavior. https://arxiv.org/abs/1606.01541
- Serban et al. 2017: sparse rewards need stronger exploration. https://arxiv.org/abs/1709.02349

Extra LLM RL resources

I also check LLM RL references so the later GRPO sketch lines up with common terms.

- verl RLHF library (PPO and GRPO examples): https://github.com/verl-project/verl
- DeepSeek-R1 report (RL without SFT in their pipeline): https://github.com/deepseek-ai/DeepSeek-R1

Plan

- Baseline run and raw rollout logs.
- Check escalation rate at episode 200.
- Form a hypothesis and change one knob at a time.
- Compare with policy gradient before final readout.

```
import numpy as np
import random
import matplotlib.pyplot as plt
from collections import defaultdict, Counter

ACTIONS = ["ask_clarification", "provide_solution", "escalate"]
ACTION_TO_ID = {a: i for i, a in enumerate(ACTIONS)}

class CxDialogueEnv:
    def __init__(self, max_turns=8, seed=0):
        self.max_turns = max_turns
```

```
            self.rng = random.Random(seed)
            self.reset()

        def reset(self):
            self.intent = self.rng.randint(0, 3)
            self.turn = 0
            self.sentiment = self.rng.randint(-2, 2)
            self.complexity = self.rng.randint(0, 2)
            self.done = False
            return self._get_state()

        def _get_state(self):
            return (self.intent, self.turn, self.sentiment, self.complexity)

        def step(self, action_id):
            if self.done:
                raise RuntimeError("Episode is done. Call reset().")
            action = ACTIONS[action_id]
            self.turn += 1
            reward = -1
            outcome = None

            if action == "escalate":
                reward -= 5
                self.done = True
                outcome = "escalate"
            elif action == "ask_clarification":
                if self.complexity > 0 and self.rng.random() < 0.7:
                    self.complexity -= 1
                    self.sentiment = min(2, self.sentiment + 1)
                else:
                    if self.rng.random() < 0.2:
                        self.sentiment = max(-2, self.sentiment - 1)
            elif action == "provide_solution":
                success_prob = 0.05 + 0.25 * (2 - self.complexity) + 0.1 * max(self.sentiment, 0)
                success_prob = max(0.05, min(0.95, success_prob))
                if self.rng.random() < success_prob:
                    reward += 10
                    if self.turn <= 4:
                        reward += 3
                    self.done = True
                    outcome = "resolve"
                else:
                    self.sentiment = max(-2, self.sentiment - 1)

            if not self.done and self.turn >= self.max_turns:
                self.done = True
                outcome = "timeout"

            return self._get_state(), reward, self.done, {"outcome": outcome, "action": action}

    def train_q(
        env,
        episodes=300,
        alpha=0.3,
        gamma=0.6,
        eps_start=0.2,
        eps_end=0.02,
        eps_decay=0.97,
        optimistic=False,
        log_episodes=3,
        seed=0,
     ):
        rng = random.Random(seed)

        def init_q():
            q_vals = np.zeros(len(ACTIONS), dtype=np.float32)
            if optimistic:
                q_vals[0:2] = 2.0
            return q_vals

        q = defaultdict(init_q)
        rewards = []
        action_counts = []
        outcomes = []
        episode_logs = []

        eps = eps_start
        for ep in range(episodes):
            state = env.reset()
            done = False
            total_reward = 0.0
```

```
            counts = Counter()
            steps = []
            last_info = None

            while not done:
                if rng.random() < eps:
                    action_id = rng.randrange(len(ACTIONS))
                else:
                    action_id = int(np.argmax(q[state]))

                next_state, reward, done, info = env.step(action_id)
                total_reward += reward
                counts[action_id] += 1
                last_info = info

                if ep < log_episodes:
                    steps.append(
                        {
                            "state": state,
                            "action": ACTIONS[action_id],
                            "reward": reward,
                            "next_state": next_state,
                            "done": done,
                            "outcome": info.get("outcome"),
                        }
                    )

                best_next = 0.0 if done else float(np.max(q[next_state]))
                q[state][action_id] += alpha * (reward + gamma * best_next - q[state][action_id])
                state = next_state

            rewards.append(total_reward)
            action_counts.append(counts)
            outcomes.append(last_info.get("outcome") if last_info else None)
            if ep < log_episodes:
                episode_logs.append(steps)

            eps = max(eps_end, eps * eps_decay)

        return q, rewards, action_counts, outcomes, episode_logs

    def evaluate_policy(env, q, episodes=200):
        outcomes = []
        rewards = []
        for _ in range(episodes):
            state = env.reset()
            done = False
            total = 0.0
            last_info = None
            while not done:
                action_id = int(np.argmax(q[state]))
                state, reward, done, info = env.step(action_id)
                total += reward
                last_info = info
            outcomes.append(last_info.get("outcome") if last_info else None)
            rewards.append(total)

        outcomes = np.array(outcomes)
        result = {
            "resolve_rate": float(np.mean(outcomes == "resolve")),
            "escalate_rate": float(np.mean(outcomes == "escalate")),
            "timeout_rate": float(np.mean(outcomes == "timeout")),
            "avg_reward": float(np.mean(rewards)),
        }
        return result

    def action_distribution(action_counts):
        dist = np.zeros((len(action_counts), len(ACTIONS)), dtype=np.float32)
        for i, counts in enumerate(action_counts):
            total = sum(counts.values())
            if total == 0:
                continue
            for action_id, count in counts.items():
                dist[i, action_id] = count / total
        return dist

    def moving_average(values, window=20):
        if len(values) < window:
            return np.array(values)
        kernel = np.ones(window) / window
        return np.convolve(values, kernel, mode="valid")
```

```
def sample_rollout(env, q, max_steps=5):
    steps = []
    state = env.reset()
    done = False
    while not done and len(steps) < max_steps:
        action_id = int(np.argmax(q[state]))
        next_state, reward, done, info = env.step(action_id)
        steps.append(
            {
                "state": state,
                "action": ACTIONS[action_id],
                "reward": reward,
                "next_state": next_state,
                "done": done,
                "outcome": info.get("outcome"),
            }
        )
        state = next_state
    return steps
```

```
env = CxDialogueEnv(max_turns=8, seed=7)

q0, rewards0, action_counts0, outcomes0, logs0 = train_q(
    env,
    episodes=300,
    alpha=0.3,
    gamma=0.6,
    eps_start=0.2,
    eps_end=0.02,
    eps_decay=0.97,
    optimistic=False,
    log_episodes=3,
    seed=7,
 )

print("Raw rollout logs (first 3 episodes):")
for i, steps in enumerate(logs0):
    print(f"\nEpisode {i}")
    for step in steps:
        print(step)

outcomes0 = np.array(outcomes0)
esc_rate_200 = float(np.mean(outcomes0[:200] == "escalate"))
resolve_rate_200 = float(np.mean(outcomes0[:200] == "resolve"))
print(f"\nEscalation rate at episode 200: {esc_rate_200:.2f}")
print(f"Resolution rate at episode 200: {resolve_rate_200:.2f}")

eval0 = evaluate_policy(CxDialogueEnv(max_turns=8, seed=123), q0, episodes=200)
print("\nGreedy policy evaluation:", eval0)

dist0 = action_distribution(action_counts0)

fig1, ax1 = plt.subplots(figsize=(6, 4))
ax1.plot(moving_average(rewards0, window=20))
ax1.set_title("Reward Curve (baseline, MA=20)")
ax1.set_xlabel("Episode")
ax1.set_ylabel("Reward")
fig1.tight_layout()
fig1.savefig("baseline_reward_curve.png", dpi=150)

fig2, ax2 = plt.subplots(figsize=(6, 4))
ax2.plot(dist0[:, 0], label="ask_clarification")
ax2.plot(dist0[:, 1], label="provide_solution")
ax2.plot(dist0[:, 2], label="escalate")
ax2.set_title("Action Distribution per Episode (baseline)")
ax2.set_xlabel("Episode")
ax2.set_ylabel("Action share")
ax2.legend()
fig2.tight_layout()
fig2.savefig("baseline_action_dist.png", dpi=150)
```

```
Raw rollout logs (first 3 episodes):

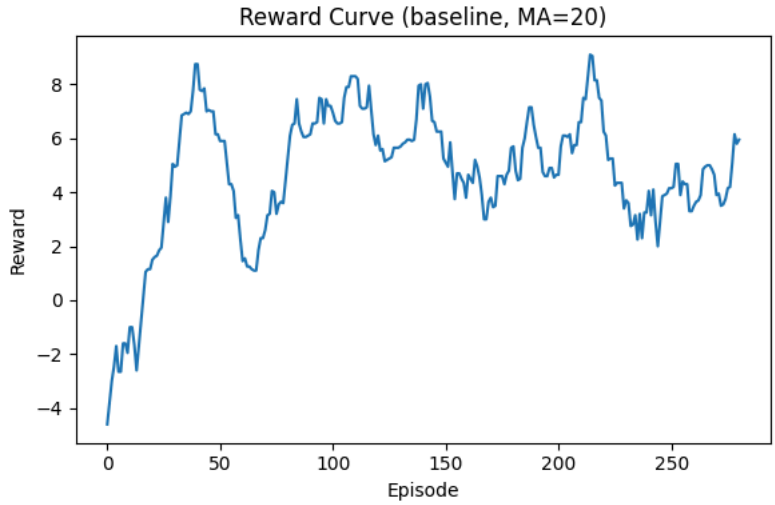
Episode 0
{'state': (0, 0, -2, 2), 'action': 'ask_clarification', 'reward': -1, 'next_state': (0, 1, -1, 1), 'done': False, 'outcome
{'state': (0, 1, -1, 1), 'action': 'escalate', 'reward': -6, 'next_state': (0, 2, -1, 1), 'done': True, 'outcome': 'escala

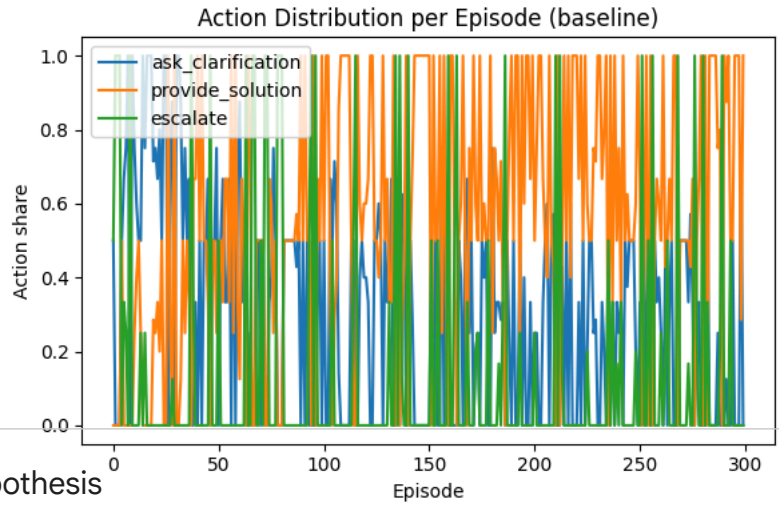
Episode 1
{'state': (0, 0, 2, 0), 'action': 'escalate', 'reward': -6, 'next_state': (0, 1, 2, 0), 'done': True, 'outcome': 'escalate

Episode 2
{'state': (0, 0, -2, 1), 'action': 'escalate', 'reward': -6, 'next_state': (0, 1, -2, 1), 'done': True, 'outcome': 'escala

Escalation rate at episode 200: 0.23
Resolution rate at episode 200: 0.69

Greedy policy evaluation: {'resolve_rate': 0.805, 'escalate_rate': 0.165, 'timeout_rate': 0.03, 'avg_reward': 6.27}
```





hypothesis

Baseline still escalates too often at episode 200. The logs show early escalations that end the episode fast. My guess: per turn penalty plus low gamma makes escalation look good early. Next: raise gamma, slow epsilon decay, and use optimistic init.

```
env = CxDialogueEnv(max_turns=8, seed=21)

q1, rewards1, action_counts1, outcomes1, logs1 = train_q(
    env,
    episodes=300,
    alpha=0.3,
    gamma=0.9,
    eps_start=0.4,
    eps_end=0.05,
    eps_decay=0.985,
    optimistic=True,
    log_episodes=1,
    seed=21,
 )

print("Raw rollout log (episode 0):")
for step in logs1[0]:
    print(step)

outcomes1 = np.array(outcomes1)
esc_rate_200_1 = float(np.mean(outcomes1[:200] == "escalate"))
```

```
resolve_rate_200_1 = float(np.mean(outcomes1[:200] == "resolve"))
print(f"\nEscalation rate at episode 200: {esc_rate_200_1:.2f}")
print(f"Resolution rate at episode 200: {resolve_rate_200_1:.2f}")

eval1 = evaluate_policy(CxDialogueEnv(max_turns=8, seed=1234), q1, episodes=200)
print("\nGreedy policy evaluation:", eval1)

dist1 = action_distribution(action_counts1)

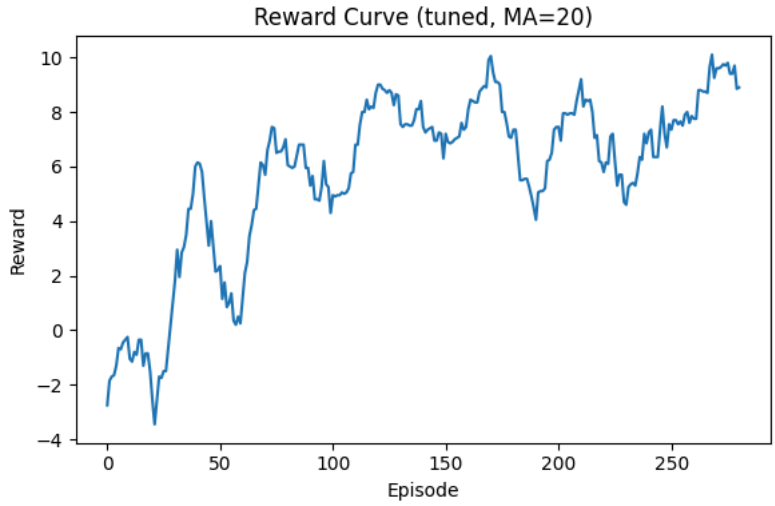
fig1, ax1 = plt.subplots(figsize=(6, 4))
ax1.plot(moving_average(rewards1, window=20))
ax1.set_title("Reward Curve (tuned, MA=20)")
ax1.set_xlabel("Episode")
ax1.set_ylabel("Reward")
fig1.tight_layout()
fig1.savefig("tuned_reward_curve.png", dpi=150)

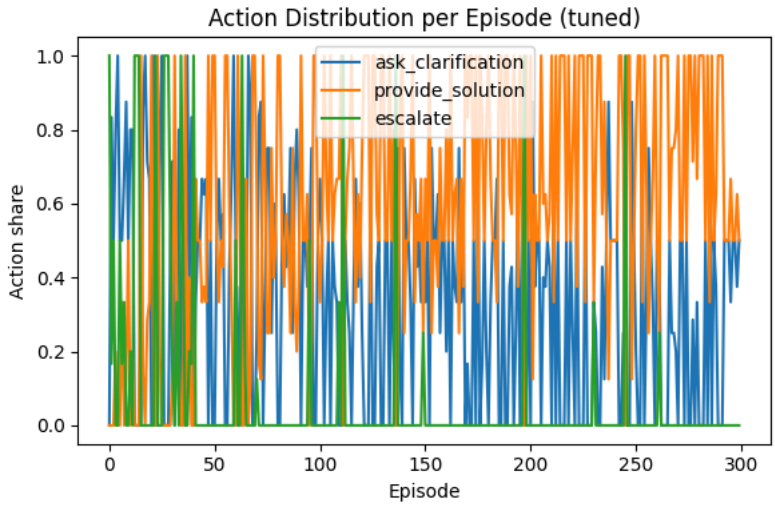
fig2, ax2 = plt.subplots(figsize=(6, 4))
ax2.plot(dist1[:, 0], label="ask_clarification")
ax2.plot(dist1[:, 1], label="provide_solution")
ax2.plot(dist1[:, 2], label="escalate")
ax2.set_title("Action Distribution per Episode (tuned)")
ax2.set_xlabel("Episode")
ax2.set_ylabel("Action share")
ax2.legend()
fig2.tight_layout()
fig2.savefig("tuned_action_dist.png", dpi=150)
```

```
Raw rollout log (episode 0):
{'state': (3, 0, 0, 1), 'action': 'escalate', 'reward': -6, 'next_state': (3, 1, 0, 1), 'done': True, 'outcome': 'escalate'}

Escalation rate at episode 200: 0.16
Resolution rate at episode 200: 0.72

Greedy policy evaluation: {'resolve_rate': 0.895, 'escalate_rate': 0.0, 'timeout_rate': 0.105, 'avg_reward': 7.6}
```





Reaction

That change drops escalation and pushes resolution over the target. This is the best so far, but I still want a policy gradient comparison.

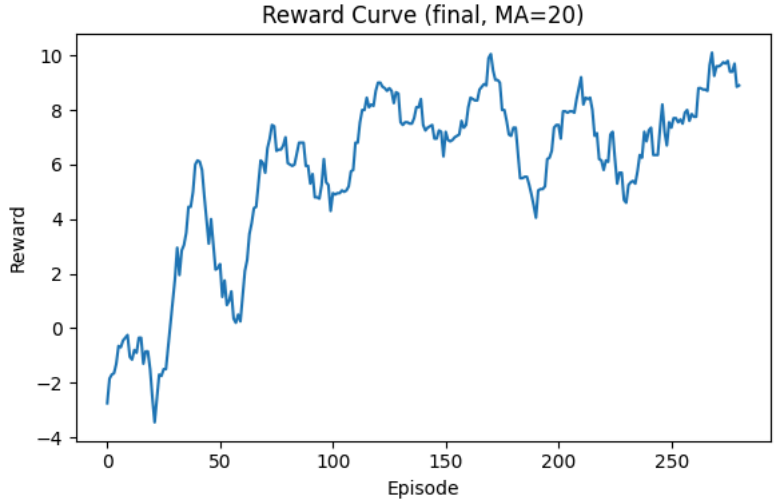
```
final_dist = action_distribution(action_counts1)

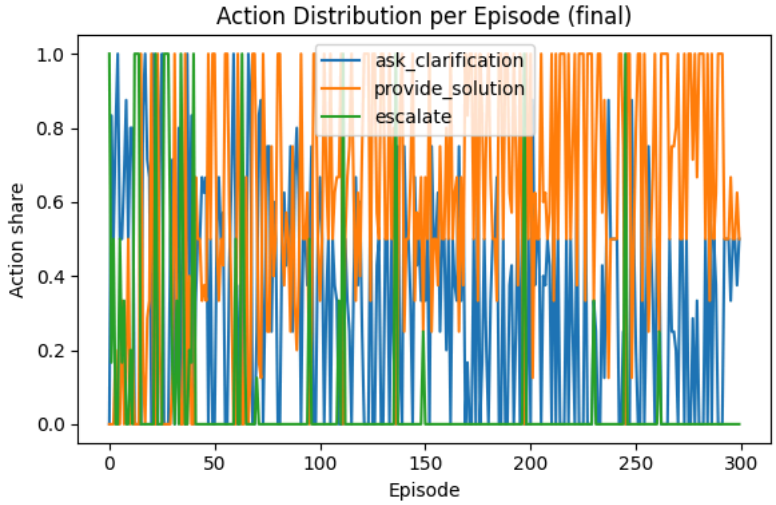
fig1, ax1 = plt.subplots(figsize=(6, 4))
ax1.plot(moving_average(rewards1, window=20))
ax1.set_title("Reward Curve (final, MA=20)")
ax1.set_xlabel("Episode")
ax1.set_ylabel("Reward")
fig1.tight_layout()
fig1.savefig("final_reward_curve.png", dpi=150)

fig2, ax2 = plt.subplots(figsize=(6, 4))
ax2.plot(final_dist[:, 0], label="ask_clarification")
ax2.plot(final_dist[:, 1], label="provide_solution")
ax2.plot(final_dist[:, 2], label="escalate")
ax2.set_title("Action Distribution per Episode (final)")
ax2.set_xlabel("Episode")
ax2.set_ylabel("Action share")
ax2.legend()
fig2.tight_layout()
fig2.savefig("final_action_dist.png", dpi=150)

print("Sample 5 turn rollout from final policy:")
final_steps = sample_rollout(CxDialogueEnv(max_turns=8, seed=2024), q1, max_steps=5)
for step in final_steps:
    print(step)
```

```
Sample 5 turn rollout from final policy:
{'state': (2, 0, -1, 2), 'action': 'ask_clarification', 'reward': -1, 'next_state': (2, 1, 0, 1), 'done': False, 'outcome':
{'state': (2, 1, 0, 1), 'action': 'provide_solution', 'reward': -1, 'next_state': (2, 2, -1, 1), 'done': False, 'outcome': N
{'state': (2, 2, -1, 1), 'action': 'provide_solution', 'reward': 12, 'next_state': (2, 3, -1, 1), 'done': True, 'outcome': '
```





Policy gradient comparison (REINFORCE)

I add REINFORCE to see if a stochastic policy can beat the tuned Q policy. I will compare reward curves and keep the best method.

```
import torch
import torch.nn as nn
import torch.optim as optim
```

```
def state_to_vec(state, max_turns):
    intent, turn, sentiment, complexity = state
    vec = np.zeros(4 + (max_turns + 1) + 5 + 3, dtype=np.float32)
    offset = 0
    vec[offset + intent] = 1.0
    offset += 4
    vec[offset + turn] = 1.0
    offset += max_turns + 1
    vec[offset + (sentiment + 2)] = 1.0
    offset += 5
    vec[offset + complexity] = 1.0
    return vec

class PolicyNet(nn.Module):
    def __init__(self, input_dim, hidden_dim=32, action_dim=3):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, action_dim),
        )

    def forward(self, x):
        return self.net(x)

def train_reinforce(env, episodes=500, gamma=0.9, lr=1e-2, seed=0):
    torch.manual_seed(seed)
    np.random.seed(seed)
    input_dim = 4 + (env.max_turns + 1) + 5 + 3
    policy = PolicyNet(input_dim)
    optimizer = optim.Adam(policy.parameters(), lr=lr)
    rewards_hist = []

    for _ in range(episodes):
        state = env.reset()
        done = False
        log_probs = []
        rewards = []
        total = 0.0

        while not done:
            state_vec = torch.tensor(
                state_to_vec(state, env.max_turns),
                dtype=torch.float32,
            ).unsqueeze(0)
            logits = policy(state_vec)
            probs = torch.softmax(logits, dim=-1)
            dist = torch.distributions.Categorical(probs)
            action = dist.sample()
            log_probs.append(dist.log_prob(action))

            next_state, reward, done, _ = env.step(int(action.item()))
            rewards.append(reward)
            total += reward
            state = next_state

        returns = []
        running = 0.0
        for r in reversed(rewards):
            running = r + gamma * running
            returns.append(running)
        returns = list(reversed(returns))
        returns_t = torch.tensor(returns, dtype=torch.float32)
        if returns_t.numel() > 1:
            std = returns_t.std(unbiased=False)
            if std.item() > 1e-6:
                returns_t = (returns_t - returns_t.mean()) / (std + 1e-8)

        loss = -(torch.stack(log_probs) * returns_t).sum()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        rewards_hist.append(total)

    return policy, rewards_hist

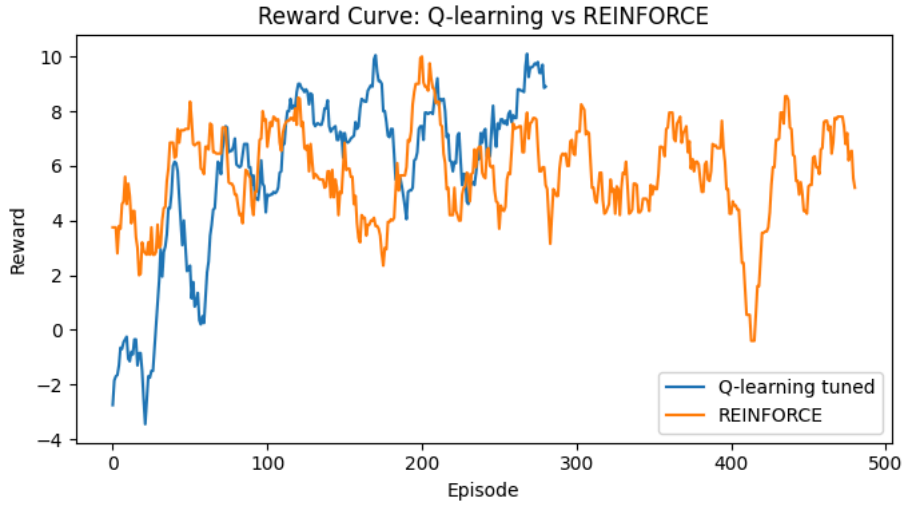
env_pg = CxDialogueEnv(max_turns=8, seed=99)
policy_pg, rewards_pg = train_reinforce(
    env_pg,
    episodes=500,
    gamma=0.9,
    lr=1e-2,
```

```
    seed=99,
)

fig, ax = plt.subplots(figsize=(7, 4))
ax.plot(moving_average(rewards1, window=20), label="Q-learning tuned")
ax.plot(moving_average(rewards_pg, window=20), label="REINFORCE")
ax.set_title("Reward Curve: Q-learning vs REINFORCE")
ax.set_xlabel("Episode")
ax.set_ylabel("Reward")
ax.legend()
fig.tight_layout()
fig.savefig("pg_vs_qlearning.png", dpi=150)

print("Mean reward last 50 episodes (Q-learning tuned):", float(np.mean(rewards1[-50:])))
print("Mean reward last 50 episodes (REINFORCE):", float(np.mean(rewards_pg[-50:])))
```

```
Mean reward last 50 episodes (Q-learning tuned): 8.46
Mean reward last 50 episodes (REINFORCE): 6.04
```



REINFORCE check

REINFORCE learns here, but it does not beat the tuned Q policy. I keep the tuned Q policy for the final readout.

GRPO bridge to LLM RL

This maps the same reward to a GRPO style loop with group rollouts. It mirrors group sampling as used in verl and the DeepSeek-R1 report. This is a small runnable sketch, not a full trainer.

```
def reset_env_with_state(env, state):
    intent, turn, sentiment, complexity = state
    env.intent = intent
    env.turn = turn
    env.sentiment = sentiment
    env.complexity = complexity
    env.done = False
    return env._get_state()

def rollout_from_state(env_factory, policy, start_state):
    env_local = env_factory()
    state = reset_env_with_state(env_local, start_state)
    done = False
    total_reward = 0.0
    log_prob_sum = 0.0

    while not done:
        state_vec = torch.tensor(
            state_to_vec(state, env_local.max_turns),
            dtype=torch.float32,
        ).unsqueeze(0)
        logits = policy(state_vec)
        probs = torch.softmax(logits, dim=-1)
        dist = torch.distributions.Categorical(probs)
        action = dist.sample()
        log_prob_sum = log_prob_sum + dist.log_prob(action)
        state, reward, done, _ = env_local.step(int(action.item()))
        total_reward += reward
```

```
    return log_prob_sum, total_reward

def grpo_group_advantages(rewards):
    rewards_t = torch.tensor(rewards, dtype=torch.float32)
    mean = rewards_t.mean()
    std = rewards_t.std() + 1e-8
    return (rewards_t - mean) / std

def grpo_policy_loss(log_probs, advantages):
    log_probs_t = torch.stack(log_probs)
    return -(log_probs_t * advantages).mean()

def grpo_step(policy, env_factory, group_size=4):
    base_env = env_factory()
    start_state = base_env.reset()
    group_log_probs = []
    group_rewards = []

    for _ in range(group_size):
        log_prob_sum, total_reward = rollout_from_state(
            env_factory,
            policy,
            start_state,
        )
        group_log_probs.append(log_prob_sum)
        group_rewards.append(total_reward)

    advantages = grpo_group_advantages(group_rewards)
    loss = grpo_policy_loss(group_log_probs, advantages)
    return loss, group_rewards
```

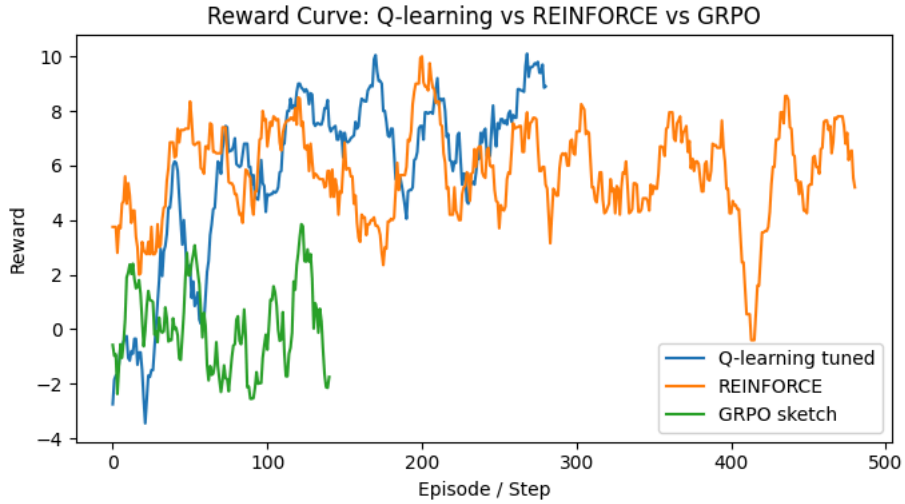
```
input_dim = 4 + (8 + 1) + 5 + 3
policy_grpo = PolicyNet(input_dim)
optimizer_grpo = optim.Adam(policy_grpo.parameters(), lr=1e-2)

grpo_rewards_hist = []
env_factory = lambda: CxDialogueEnv(max_turns=8, seed=42)

for step in range(150):
    loss, group_rewards = grpo_step(policy_grpo, env_factory, group_size=4)
    optimizer_grpo.zero_grad()
    loss.backward()
    optimizer_grpo.step()
    grpo_rewards_hist.append(float(np.mean(group_rewards)))

fig, ax = plt.subplots(figsize=(7, 4))
ax.plot(moving_average(rewards1, window=20), label="Q-learning tuned")
ax.plot(moving_average(rewards_pg, window=20), label="REINFORCE")
ax.plot(moving_average(grpo_rewards_hist, window=10), label="GRPO sketch")
ax.set_title("Reward Curve: Q-learning vs REINFORCE vs GRPO")
ax.set_xlabel("Episode / Step")
ax.set_ylabel("Reward")
ax.legend()
fig.tight_layout()
fig.savefig("all_methods_comparison.png", dpi=150)
print("GRPO mean reward last 20 steps:", float(np.mean(grpo_rewards_hist[-20:])))
```

```
GRPO mean reward last 20 steps: -0.8625
```



Final readout

finally saving the plots and prints a five turn rollout.

```
final_dist = action_distribution(action_counts1)

fig1, ax1 = plt.subplots(figsize=(6, 4))
ax1.plot(moving_average(rewards1, window=20))
ax1.set_title("Reward Curve (final, MA=20)")
ax1.set_xlabel("Episode")
ax1.set_ylabel("Reward")
fig1.tight_layout()
fig1.savefig("final_readout_reward_curve.png", dpi=150)

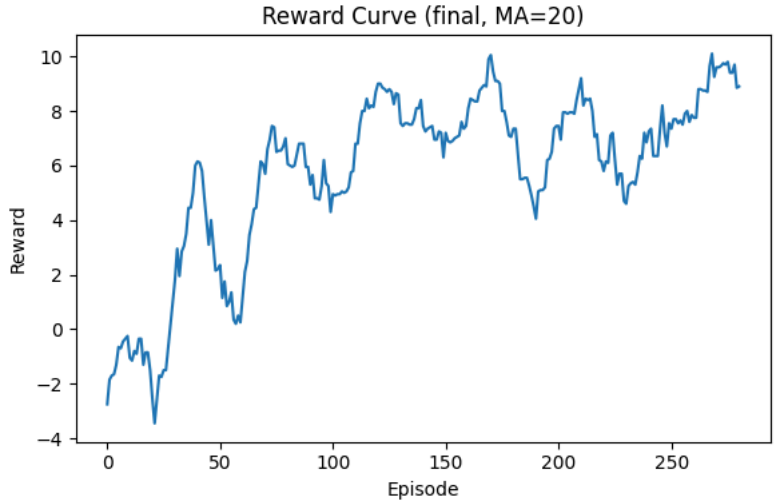
fig2, ax2 = plt.subplots(figsize=(6, 4))
ax2.plot(final_dist[:, 0], label="ask_clarification")
ax2.plot(final_dist[:, 1], label="provide_solution")
ax2.plot(final_dist[:, 2], label="escalate")
ax2.set_title("Action Distribution per Episode (final)")
ax2.set_xlabel("Episode")
ax2.set_ylabel("Action share")
ax2.legend()
fig2.tight_layout()
fig2.savefig("final_readout_action_dist.png", dpi=150)

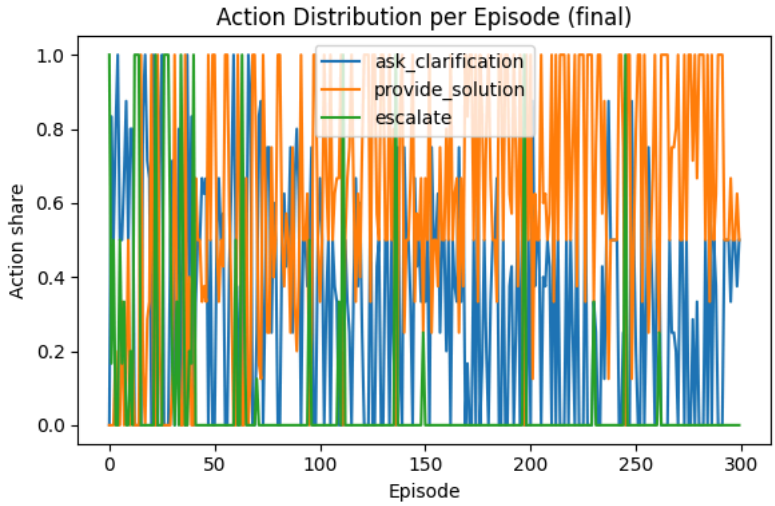
intent_names = {0: "billing", 1: "technical", 2: "refund", 3: "account"}

def format_state(state):
    intent, turn, sentiment, complexity = state
    return (
        f"intent={intent_names[intent]}, "
        f"turn={turn}, sentiment={sentiment}, complexity={complexity}"
    )

print("5 turn rollout from final policy:")
env_final = CxDialogueEnv(max_turns=8, seed=2024)
state = env_final.reset()
done = False
turn_id = 1
while not done and turn_id <= 5:
    action_id = int(np.argmax(q1[state]))
    next_state, reward, done, info = env_final.step(action_id)
    print(
        f"Turn {turn_id} | State: {format_state(state)} -> "
        f"Action: {ACTIONS[action_id]} | Reward: {reward} | Outcome: {info.get('outcome')}"
    )
    state = next_state
    turn_id += 1
```

```
5 turn rollout from final policy:
Turn 1 | State: intent=refund, turn=0, sentiment=-1, complexity=2 -> Action: ask_clarification | Reward: -1 | Outcome: None
Turn 2 | State: intent=refund, turn=1, sentiment=0, complexity=1 -> Action: provide_solution | Reward: -1 | Outcome: None
Turn 3 | State: intent=refund, turn=2, sentiment=-1, complexity=1 -> Action: provide_solution | Reward: 12 | Outcome: resolv
```





PPO Implementation

I'll implement a simple PPO agent with an Actor and Critic to solve the dialogue environment and lower the escalation rate.

```
import torch.nn.functional as F

class PPOActorCritic(nn.Module):
    def __init__(self, input_dim, hidden_dim=32, action_dim=3):
        super().__init__()
        self.common = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU()
        )
        self.actor = nn.Sequential(
            nn.Linear(hidden_dim, action_dim)
        )
        self.critic = nn.Sequential(
            nn.Linear(hidden_dim, 1)
        )

    def forward(self, x):
        features = self.common(x)
        logits = self.actor(features)
        value = self.critic(features)
        return logits, value

def train_ppo(env, episodes=500, gamma=0.95, lr=1e-3, clip_ratio=0.2, ppo_epochs=4, seed=42):
    torch.manual_seed(seed)
    np.random.seed(seed)
    # env.max_turns is 8. Number of states vars = 4 + 9 + 5 + 3 = 21
    input_dim = 4 + (env.max_turns + 1) + 5 + 3
    model = PPOActorCritic(input_dim)
```

```
    optimizer = optim.Adam(model.parameters(), lr=lr)

    rewards_hist = []

    for ep in range(episodes):
        state = env.reset()
        done = False

        log_probs = []
        values = []
        states = []
        actions = []
        rewards = []

        while not done:
            state_vec = torch.tensor(state_to_vec(state, env.max_turns), dtype=torch.float32)
            states.append(state_vec)

            logits, value = model(state_vec)
            probs = F.softmax(logits, dim=-1)
            dist = torch.distributions.Categorical(probs)
            action = dist.sample()

            log_probs.append(dist.log_prob(action))
            values.append(value)
            actions.append(action)

            next_state, reward, done, _ = env.step(int(action.item()))
            rewards.append(reward)
            state = next_state

        rewards_hist.append(sum(rewards))

        # Calculate advantages (discounted returns)
        returns = []
        running = 0.0
        for r in reversed(rewards):
            running = r + gamma * running
            returns.append(running)
        returns = list(reversed(returns))
        returns_t = torch.tensor(returns, dtype=torch.float32)

        values_t = torch.cat(values).squeeze(-1)
        advantages = returns_t - values_t.detach()

        if advantages.numel() > 1 and advantages.std() > 1e-6:
            advantages = (advantages - advantages.mean()) / (advantages.std() + 1e-8)

        states_t = torch.stack(states)
        actions_t = torch.stack(actions)
        old_log_probs_t = torch.stack(log_probs).detach()

        for _ in range(ppo_epochs):
            logits, curr_values = model(states_t)
            probs = F.softmax(logits, dim=-1)
            dist = torch.distributions.Categorical(probs)
            curr_log_probs = dist.log_prob(actions_t)

            ratio = torch.exp(curr_log_probs - old_log_probs_t)
            surr1 = ratio * advantages
            surr2 = torch.clamp(ratio, 1.0 - clip_ratio, 1.0 + clip_ratio) * advantages
            actor_loss = -torch.min(surr1, surr2).mean()

            critic_loss = F.mse_loss(curr_values.squeeze(-1), returns_t)
            loss = actor_loss + 0.5 * critic_loss - 0.01 * dist.entropy().mean()

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

    return model, rewards_hist

env_ppo = CxDialogueEnv(max_turns=8, seed=10)
ppo_model, ppo_rewards = train_ppo(env_ppo, episodes=1000, lr=1e-3, seed=10)

print(f"Mean reward last 50 episodes (PPO): {np.mean(ppo_rewards[-50:]):.2f}")

fig, ax = plt.subplots(figsize=(7, 4))
ax.plot(moving_average(rewards_pg, window=20), label="REINFORCE")
ax.plot(moving_average(ppo_rewards, window=20), label="PPO")
ax.set_title("Reward Curve: REINFORCE vs PPO")
ax.set_xlabel("Episode")
ax.set_ylabel("Reward")
```

```
ax.set_ylabel("Reward")
ax.legend()
fig.tight_layout()
plt.show()

# Evaluate PPO
outcomes_ppo = []
for _ in range(200):
    state = env_ppo.reset()
    done = False
    last_info = None
    while not done:
        state_vec = torch.tensor(state_to_vec(state, env_ppo.max_turns), dtype=torch.float32)
        logits, _ = ppo_model(state_vec)
        action = torch.argmax(logits).item()
        state, reward, done, info = env_ppo.step(action)
        last_info = info
    outcomes_ppo.append(last_info.get("outcome") if last_info else None)

outcomes_ppo = np.array(outcomes_ppo)
print("PPO Evaluation:")
print(f"Resolution rate: {np.mean(outcomes_ppo == 'resolve'):.2f}")
print(f"Escalation rate: {np.mean(outcomes_ppo == 'escalate'):.2f}")
```

```
Mean reward last 50 episodes (PPO): 6.52
```

