Causal Reinforcement Learning

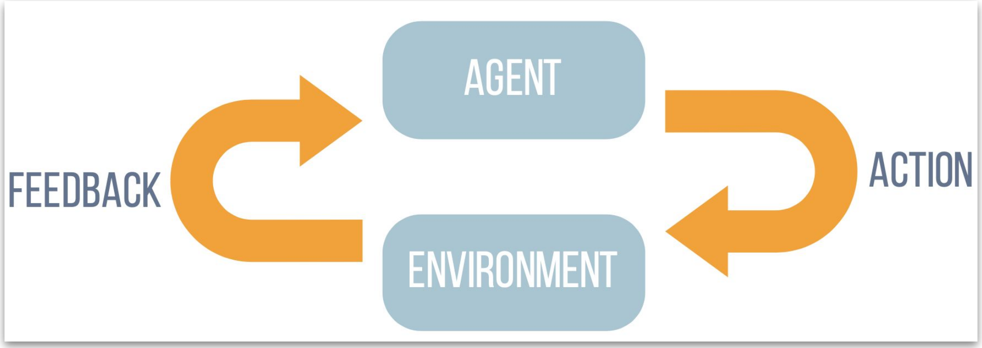
By:
Andrea Baisero
Prakhar Patidar
Rishabh Shanbhag
Sagar Singh

Multi-track project, contributions:

- Planning as inference:
  - Working pseudo-softmax agent capable of solving FrozenLake (with minimal reward shaping and no knowledge of the environment).
  - Non-so-much-working other pseudo-softmax implementations.
- Generalization to other environments:
  - Parsers for standard MDP and POMDP formats.
  - PyroMDP & PyroPOMDP, OpenAI Gym environments which run as pyro probabilistic programs.
  - Working softmax agent capable of solving \`gridworld.mdp\` environment.
- Preliminary study on confounding MDPs:
  - Novel CMDP format for MDPs with static confounding variables.
  - PyroCMDP, OpenAI Gym environment which runs as a pyro probabilistic program.
  - Explored difference between “conditional” RL and causal RL on \`circle.cmdp\` environment.

Reinforcement Learning: Intuition

These are set of algorithms that are best suited for solving sequential decision making problem, with the end goal of finding the right balance between reward and exploration.



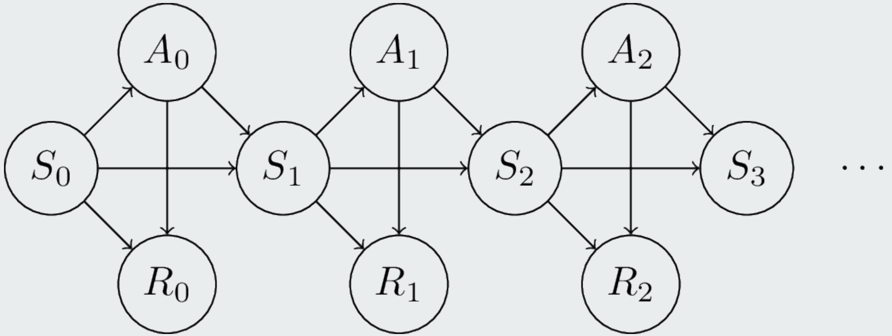
Foreword: What even is RL?

No single “RL problem”, many variants:

- Fully observable VS partially observable
- Finite horizon VS infinite horizon VS indefinite horizon
- Episodic VS continuing
- Evaluation VS learning VS control
- Model-free VS model-based
- Policy-based methods VS Value-based methods
- On-policy VS off-policy
- Single agent VS multi agent
- Bayesian RL, Inverse RL, Control as inference, Max-Entropy RL, Causal RL, etc...

$\Longrightarrow$ no such thing as **the** RL problem, VERY important to be clear about exact problem framing and assumptions.

MDPs: Markov Decision Processes



$$\Pr(S_{t+1} \mid S_0, \ldots, S_t, A_t) = \Pr(S_{t+1} \mid S_t, A_t)$$

An MDP is a tuple $\langle S, A, T, R, \gamma \rangle$

- State space S
- Action space A
- Transition function $T \colon S \times A \to \Delta S$
- Reward function $R \colon S \times A \to \mathbb{R}$
- Discount factor $\gamma \in (0, 1]$

An agent is represented by a <u>policy</u>

- Policy function $\pi \colon S \to \Delta A$

The optimal policy maximizes <u>expected return</u>

- Return $G = \sum_{t=0}^{\infty} \gamma^t R(S_t, A_t)$
- Action values $Q^\pi(s, a) \mapsto \mathbb{E}_\pi \left[G \mid S = s, A = a \right]$
- State values $V^\pi(s) = \mathbb{E}_{a \sim \pi(s)} \left[Q^\pi(s, a) \right]$
- Optimal policy $\pi^* \doteq \arg\max_{\pi \in \Pi} V^\pi(s)$

$$\pi^*(s) \mapsto \arg\max_{a \in A} \max_{\pi \in \Pi} \mathbb{E}_\pi \left[G \mid S_0 = s, A_0 = a \right]$$

Why introduce Causality in RL ?

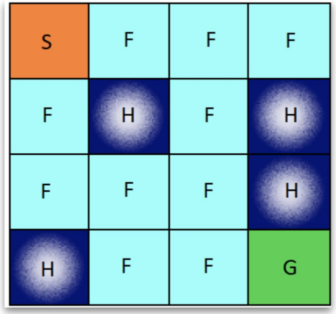
- Concept of intervention in causal inference is used to exploit the concept of action in RL.
- Assists RL in learning value functions or policies more efficiently.
- Causal inference in RL, allows RL to be able to infer causal effects between data in the complicated real-world problems.
- Helps to estimate the practical effect of treatment predicated on the existence of unobserved confounders.

Reward	state, action $\rightarrow$ reward
Transition	state, action $\rightarrow$ state
Hidden state	hidden state $\rightarrow$ observation

Table 1: Summary of causal relationships in reinforcement learning.

Environment: Frozen Lake

The Frozen Lake environment is a 4×4 grid with four possible areas — Start (S), Frozen (F), Hole (H) and Goal (G).



- The agent moves around the grid until it reaches the goal or a hole.
  - If it reaches the goal, it gets reward 1.
  - If it falls into a hole, it gets reward 0.
- The process continues until it learns from every mistake and reaches the goal eventually.

Control as Inference

Model-based RL

- Collect experience trajectories $\tau = s_0, a_0, s_1, a_1, s_2, a_2, \ldots$
- Learn a parameterized transition model $\hat{T}(s' \mid s, a; \theta)$
- Plan using the learned model $\pi(s) = \text{planner}(\hat{T}, s)$

Model-free RL

- Collect experience trajectories $\tau = s_0, a_0, s_1, a_1, s_2, a_2, \ldots$
- Optimize a parameterized policy model $\pi(s; \theta)$ usually via the policy gradient $\nabla_\theta \mathbb{E}_\pi [G]$
  OR a parameterized action-value model $Q^*(s, a; \theta)$

Control as Inference

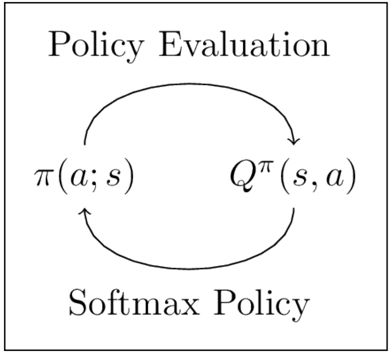
- No parametric policy, requires “prior” agent as uninformed stochastic model $\pi(a;s) = \Pr(A_0 = a \mid S_0 = s)$ e.g. uniform policy.
- “Optimal” policy expressed as an inference problem $\pi(a;s) = \Pr(A_0 = a \mid S_0 = s, \text{high } G)$

Problems:

- “High G” is problem dependent, possible or optimal returns not necessarily known.
- Solving the inference problem via sampling-based inference $\equiv$Random search + filtering (not very efficient).

Our goal was to implement a related type of agent, the **softmax agent**.

Softmax Agent



Normally, $Q^\pi(s,a) \doteq \mathbb{E}_\pi[G \mid S_0 = s, A_0 = a]$ (inference) is a function of $\pi(a;s)$.

The softmax agent “closes the loop”: $\pi(a;s)$ is a function of its own $Q^\pi(s,a)$

$$\pi(a;s) \propto \exp\left(\alpha Q^\pi(s,a)\right)$$

Properties:

- policy is a soft-max over action-values
- $\alpha \in \mathbb{R}^+$ modulates the agent’s stochasticity, can be used for exploration:
  - $\lim_{\alpha \to 0^+} \pi(s) = \mathrm{Uniform}(A)$
  - $\lim_{\alpha \to \infty} \pi(s) = \arg\max_{a \in A} Q^\pi(s,a) = \pi^*(s)$

Softmax Implementation (attempt 1/4)

`control_as_inference.py frozenlake --policy control-as-inference-like`

- First attempt at softmax!
  - Implemented for FrozenLake and PyroMDPs (discussed later).
  - Sample random trajectory, use `pyro.factor(` αG`)` to influence trace log-likelihood
  - Use importance sampling to sample action site A_0
  - Works, finds optimal policy!

- Interpretation:
  - `pyro.factor` as soft-conditioning/filtering
    $\Rightarrow$ select random trajectories which result in high sample return
  - We did not actually implement softmax, we implemented ~ control as inference!

Softmax Implementation (attempt 2/4)

```
control_as_inference.py frozenlake --policy softmax-like
```

- Second attempt at softmax!
  - Previous implementation used sample returns rather than expected returns, let’s fix that.
  - Implemented for FrozenLake and PyroMDPs (discussed later).
  - For every action $a \in \mathcal{A}$
    - Sample random trajectories
    - Use `ImportanceSampling` to estimate $Q(s, a) = \mathbb{E}\left[G \mid S_0 = s, A_0 = a\right]$
    - Explicitly compute the softmax policy from the Q-values, and sample from it.
  - Does not work, does not find optimal policy!

- Interpretation:
  - We are estimating Q^{π} where π is the prior uniform policy, not the softmax policy!
  - We did not actually implement softmax, we implemented ~ one step of generalized policy improvement.
  - We realize our first implementation, while working, is not softmax:
    - The real softmax would not sample A_1, A_2, ..., randomly!

Softmax Implementation (attempt 3/4)

softmax_recursive.py ../gridworld.mdp

- Third attempt at real softmax!
  - The real softmax will have to evaluate its own policy to compute its policy,
  - Recursive calls until time-limit is reached between:
    - `softmax_agent_model` (uses `infer_Q` to compute π)
    - `infer_Q` (uses `softmax_agent_model` to compute Q^π)
  - Does it work? Who knows!
    - Too computationally expensive to run any significant experiment.

- Possible improvements:
  - Possible to memoize intermediate results so that policy and action values are not recomputed for the same states.
  - Instead of random sampling, discrete enumeration may help computational efficiency
    - (could not get discrete enumeration working on Pyro)

Softmax Implementation (attempt 4/4)

softmax_presample_policy.py ../gridworld.mdp

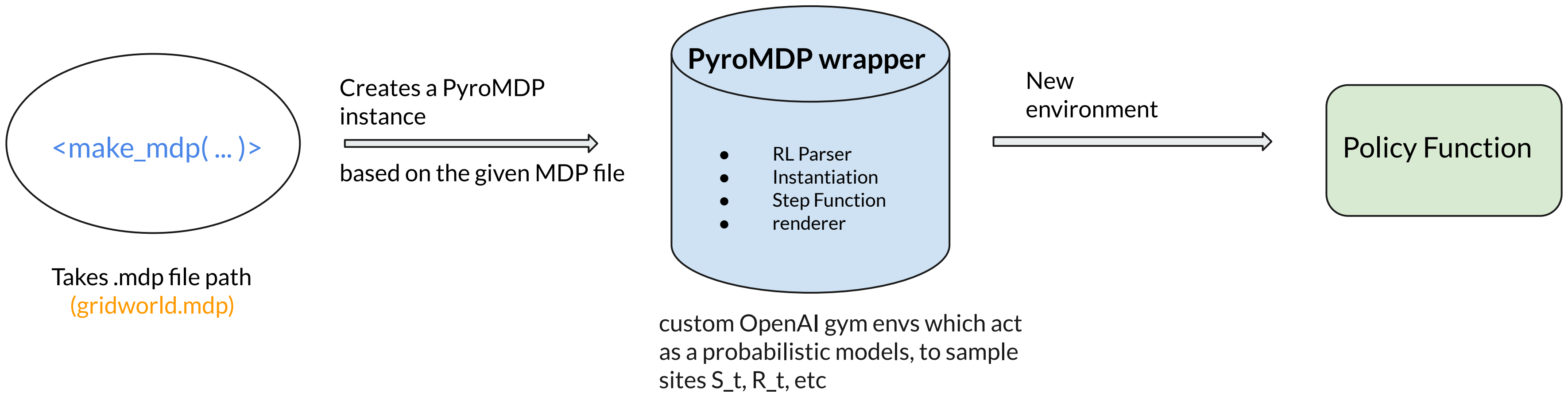
- Fourth attempt at real (and efficient) softmax!
  - Last minute addition to improve efficiency
  - We avoid the recursive calls:
    - We include deterministic policies as RVs to the model $\pi \sim \Pi \equiv A^{|S|}$
    - We estimate the policy's values Q^{π} using importance sampling
    - We use `pyro.factor(` αQ^{π} `)` to induce a preference towards policies with high
    - We sample the policy using importance sampling on the entire above process
    - We choose the action using the sample policy
  - Kinda works, sometimes! But very sensitive to hyper-parameters!

- Future work:
  - Not 100% sure this is equivalent to real softmax; requires proof.

Generalization to other environments

Generalized Agent: Workflow

Goal was to generalise the model built for Frozen Lake environment to work on different environments

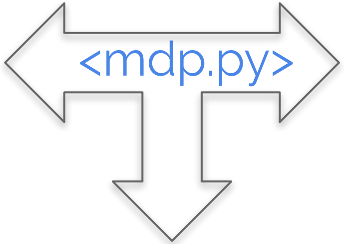


PyroMDP: OpenAI Gym environment which runs as a pyro probabilistic program

```python
def __init__(self, text, *, episodic, seed=None):
    self.model = parse(text)
    self.episodic = episodic
    self.seed(seed)

    if self.model.values == 'cost':
        raise ValueError('Unsupported `cost` values')

    self.discount = self.model.discount
    self.state_space = spaces.Discrete(len(self.model.states))
    self.action_space = spaces.Discrete(len(self.model.actions))
    self.reward_range = self.model.R.min(), self.model.R.max()

    if self.model.start is None:
        self.start = torch.ones(self.state_space.n) / self.state_space.n
    else:
        self.start = torch.from_numpy(self.model.start.copy())
    self.T = torch.from_numpy(self.model.T.transpose(1, 0, 2).copy())
    self.R = torch.from_numpy(self.model.R.transpose(1, 0, 2).copy())

    self.D = None
    if episodic:
        # only if episodic
        self.D = torch.from_numpy(self.model.reset.T.copy())

    self.__time_step = None
    self.state = None
    self.done = None
    self.action_prev = None
```

File parser

Instantiation




```python
def step(self, action):
    assert self.__time_step >= 0
    assert 0 <= self.state < self.state_space.n

    if not 0 <= action < self.action_space.n:
        raise ValueError(
            f'Action should be an integer in {{0, ..., {self.action_space.n}}}'
        )

    if self.done is None or self.__time_step is None:
        raise InternalStateError(
            'The environment must be reset before being used'
        )

    if self.done:
        raise InternalStateError(
            'The previous episode has ended and the environment must reset'
        )

    self.__time_step += 1

    state_next_dist = Categorical(self.T[self.state, action])
    state_next = pyro.sample(f'S_{self.__time_step}', state_next_dist)

    reward_dist = Delta(self.R[self.state, action, state_next])
    reward = pyro.sample(f'R_{self.__time_step}', reward_dist)

    if self.episodic:
        done = self.D[self.state, action]
    else:
        done = torch.tensor(False)

    done_probs = torch.eye(2)[done.long()]
    done_dist = Categorical(done_probs)
    done = pyro.sample(f'D_{self.__time_step}', done_dist)

    info = {
        'T': self.T[self.state, action],  # transition distribution
        'R': self.R[self.state, action],  # stochastic rewards
    }

    self.state = state_next
    self.action_prev = action

    return state_next, reward, done, info
```

<step(...)>

<render(...)>

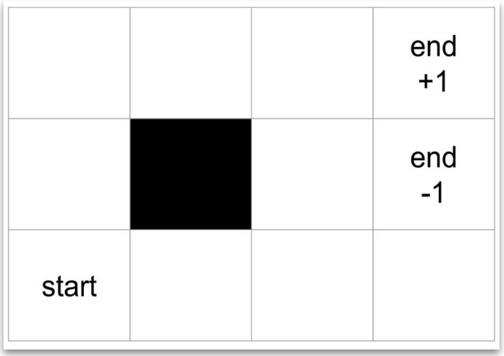
```python
def render(  # pylint: disable=inconsistent-return-statements
    self, mode='human'
):
    if mode not in ('human', 'ansi'):
        raise ValueError('Only `human` and `ansi` modes are supported')

    # stream where to send the string representation of the env
    outfile = sys.stdout if mode == 'human' else io.StringIO()

    if self.action_prev is not None:
        ai = self.action_prev.item()
        print(f'action: {self.model.actions[ai]} (#{ai})', file=outfile)

    si = self.state.item()
    print(f'state: {self.model.states[si]} (#{si})', file=outfile)

    if mode == 'ansi':
        with contextlib.closing(outfile):
            return outfile.getvalue()
```

RL Parsers

1. Contains parsers for file formats related to RL such as MDP and POMDP
2. In each case, the contents of the parsed file is returned as a namedtuple containing the fields specified by the respective file format.
3. Addition of the reset keyword, which may be used both to indicate the end of an episode in episodic tasks, and the reinitialization of the state according to the initial distribution in continuing tasks.

Gridworld environment

Agent movements are stochastic: 80% of moving in correct direction, and 20% evenly split across 2 orthogonal directions



Gridworld

When moving OUT, the reward is **-0.1**

Wall: the agent cannot move here
End + : positive terminal state; when moving OUT, the reward is **+1.0**
End - : negative terminal state; when moving OUT, the reward is **-1.0**

discount: 0.95
values: reward
states: 11
actions: north south east west
start: 7

Policy Selection: Control as Inference

```python
def main():
    assert args.policy in ('control-as-inference-like', 'softmax-like')

    if args.policy == 'control-as-inference-like':
        policy = policy_control_as_inference_like
    elif args.policy == 'softmax-like':
        policy = softmax_like

    if args.mdp == 'frozenlake':
        env = gym.make('FrozenLake-v0', is_slippery=False)
        env = FrozenLakeWrapper(env)

        trajectory_model = trajectory_model_frozenlake
        agent_model = agent_models.get_agent_model('FrozenLake-v0')

        # makes sure integer action is sent to frozenlake environment
        def action_cast(action):
            return action.item()

    else:
        env = make_mdp(args.mdp, episodic=True)
        env = TimeLimit(env, 100)

        trajectory_model = trajectory_model_mdp
        agent_model = agent_models.get_agent_model(args.mdp)

        # makes sure tensor action is sent to MDP environment
        def action_cast(action):
            return action

    env.reset()
    for t in itt.count():
        print('---')
        print(f't: {t}')
        print('state:')
        env.render()

        action = policy(
            env,
            trajectory_model=trajectory_model,
            agent_model=agent_model,
            log=True,
        )
        _, reward, done, _ = env.step(action_cast(action))
        print(f'reward: {reward}')

        if done:
            print('final state:')
            env.render()
            print(f'Episode finished after {t+1} timesteps')
            break

    env.close()
```

Passing mdp (here gridworld.mdp) as argument

```python
def trajectory_model_mdp(env, *, agent_model=None, factor_G=False):
    """trajectory_model_mdp

    A probabilistic program for MDP environment trajectories.  The sample return
    can be used to affect the trace likelihood.

    :param env: OpenAI Gym environment
    :param agent_model: agent's probabilistic program
    :param factor_G: boolean; if True then apply $\\alpha G$ likelihood factor
    """
    if agent_model is None:
        agent_model = agent_models.uniform

    env = deepcopy(env)

    # running return and discount factor
    return_, discount = 0.0, 1.0

    # with keep_state=True only the time-step used to name sites is being reset
    state = env.reset(keep_state=True)
    for t in itt.count():
        action = agent_model(f'A_{t}', env, state)
        state, reward, done, _ = env.step(action)

        # running return and discount factor
        return_ += discount * reward
        discount *= args.gamma

        if done:
            break

    pyro.sample('G', Delta(return_))

    if factor_G:
        pyro.factor('factor_G', args.alpha * return_)

    return return_
```

```python
def policy_control_as_inference_like(
    env, *, trajectory_model, agent_model, log=False
):
    """policy_control_as_inference_like

    Implements a control-as-inference-like policy which "maximizes"
    $\\Pr(A_0 \\mid S_0, high G)$.

    Not actually standard CaI, because we don't really condition on G;  rather,
    we use $\\alpha G$ as a likelihood factor on sample traces.

    :param env: OpenAI Gym environment
    :param trajectory_model: trajectory probabilistic program
    :param agent_model: agent's probabilistic program
    :param log: boolean; if True, print log info
    """
    inference = Importance(trajectory_model, num_samples=args.num_samples)
    posterior = inference.run(env, agent_model=agent_model, factor_G=True)
    marginal = EmpiricalMarginal(posterior, 'A_0')

    if log:
        samples = marginal.sample((args.num_samples,))
        counts = Counter(samples.tolist())
        hist = [counts[i] / args.num_samples for i in range(env.action_space.n)]
        print('policy:')
        print(tabulate([hist], headers=env.actions, tablefmt='fancy_grid'))

    return marginal.sample()
```

< control_as_inference.py >

Policy Selection: Softmax

Passing mdp (here gridworld.mdp) as argument

```python
def main():
    env = make_mdp(args.mdp, episodic=True)
    env = TimeLimit(env, 10)

    env.reset()
    for t in itt.count():
        print('---')
        print(f't: {t}')
        print('state:')
        env.render()

        action = policy(env, log=True)
        _, reward, done, _ = env.step(action)
        print(f'reward: {reward}')

        if done:
            print('final state:')
            env.render()
            print(f'Episode finished after {t+1} timesteps')
            break

    env.close()


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('mdp', help='path to MDP file')
    parser.add_argument('--alpha', type=float, default=5_000.0)
    parser.add_argument('--gamma', type=float, default=0.95)
    parser.add_argument('--num-samples', type=int, default=20)
    args = parser.parse_args()

    print(f'args: {args}')
    main()
```

```python
def policy(env, log=False):
    """policy

    :param env: OpenAI Gym environment
    :param log: boolean; if True, print log info
    """
    inference = Importance(softmax_agent_model, num_samples=args.num_samples)
    posterior = inference.run(env)
    marginal = EmpiricalMarginal(posterior, 'policy_vector')

    if log:
        policy_samples = marginal.sample((args.num_samples,))
        action_samples = policy_samples[:, env.state]
        counts = Counter(action_samples.tolist())
        hist = [counts[i] / args.num_samples for i in range(env.action_space.n)]
        print('policy:')
        print(tabulate([hist], headers=env.actions, tablefmt='fancy_grid'))

    policy_vector = marginal.sample()
    return policy_vector[env.state]
```

```python
def softmax_agent_model(env):
    """softmax_agent_model

    Softmax agent model;  Performs inference to estimate $Q^\pi(s, a)$, then
    uses pyro.factor to modify the trace log-likelihood.

    :param env: OpenAI Gym environment
    """
    policy_probs = torch.ones(env.state_space.n, env.action_space.n)
    policy_vector = pyro.sample('policy_vector', Categorical(policy_probs))

    inference = Importance(trajectory_model, num_samples=args.num_samples)
    posterior = inference.run(env, lambda state: policy_vector[state])
    Q = EmpiricalMarginal(posterior, 'G').mean

    pyro.factor('factor_Q', args.alpha * Q)

    return policy_vector
```

<softmax_presample_policy.py >

Results

Policy: Control as Inference

```
---
t: 0
state:
...+
. .-
....
policy:
```

north	south	east	west
0	0	0	1

```
reward: -0.1
---
t: 1
state:
action: west
...+
. .-
....
policy:
```

north	south	east	west
0.7465	0	0.2535	0

```
reward: -0.1
---
t: 2
state:
action: north
...+
. .-
....
policy:
```

north	south	east	west
0.7515	0	0.128	0.1205

```
reward: -0.1
---
```

```
t: 3
state:
action: north
...+
. .-
....
policy:
```

north	south	east	west
0.09	0.0895	0.8205	0

```
reward: -0.1
---
t: 4
state:
action: east
...+
. .-
....
policy:
```

north	south	east	west
0.0585	0.153	0.7885	0

```
reward: -0.1
---
t: 5
state:
action: east
...+
. .-
....
policy:
```

north	south	east	west
0.102	0.115	0.783	0

```
reward: -0.1
---
```

```
t: 6
state:
action: east
...+
. .-
....
policy:
```

north	south	east	west
0.2525	0.248	0.251	0.2485

```
reward: 1.0
final state:
action: south
...+
. .-
....
Episode finished after 7 timesteps
```

Policy: Softmax

```
t: 0
state:
...+
. .-
....
policy:
```

north	south	east	west
0.35	0.35	0.2	0.1

```
reward: -0.1
---
t: 1
state:
action: north
...+
. .-
....
policy:
```

north	south	east	west
0	0	1	0

```
reward: -0.1
---
t: 2
state:
action: east
...+
. .-
....
policy:
```

north	south	east	west
1	0	0	0

```
reward: -0.1
---
```

```
t: 3
state:
action: north
...+
. .-
....
policy:
```

north	south	east	west
0	0	1	0

```
reward: -0.1
---
t: 4
state:
action: east
...+
. .-
....
policy:
```

north	south	east	west
0	0	1	0

```
reward: -0.1
---
t: 5
state:
action: east
...+
. .-
....
policy:
```

north	south	east	west
0	0	1	0

```
reward: -0.1
---
```

```
t: 6
state:
action: east
...+
. .-
....
policy:
```

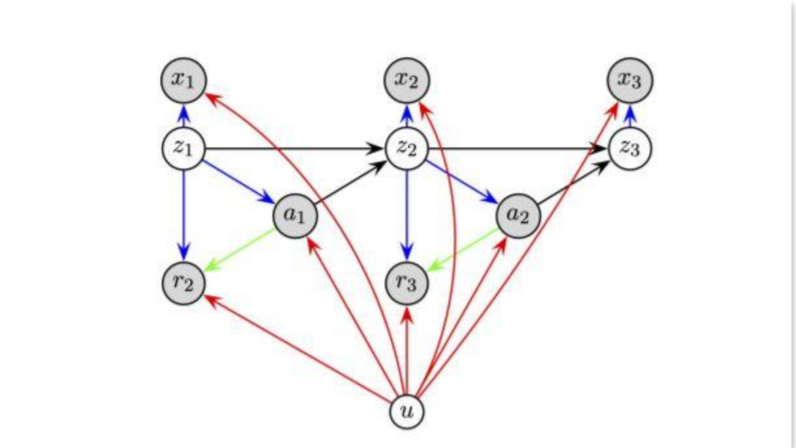
north	south	east	west
0.25	0.15	0.15	0.45

```
reward: 1.0
final state:
action: south
...+
. .-
....
Episode finished after 7 timesteps
```

Confounding MDPs

CMDP: Confounding MDP

- We created a Novel CMDP format for MDPs with static confounding variables.
- CMDPs are a special case of partially observable MDPs (POMDPs)

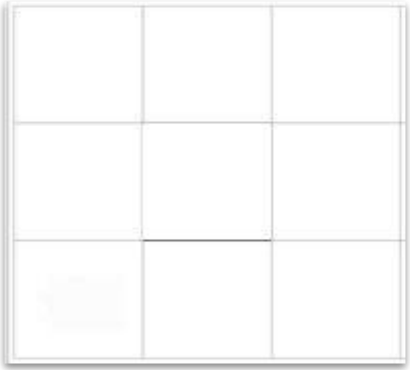


- Grey nodes here denote the observed variables
- White nodes represent the unobserved variables.
- Variables:
  a *- action*
  r *- reward*
  z *- state (directly observed in this case)*
  u *- latent confounder*
- *Green lines are the causal effects of interest*

$$\mathbb{E}\left[R_t \mid S_t = s, do(A_t = a)\right] \neq \mathbb{E}\left[R_t \mid S_t = s, A_t = a\right]$$

Circles environment

Agent movements: Depending on the confounder, the agent will receive positive rewards for either moving clockwise or counterclockwise along the border.



Circle Gridworld

discount: 0.95
values: reward
states: s00 s01 s02 s10 s11 s12 s20 s21 s22
actions: north south east west
confounders: cw ccw
start: random

- Rewards are defined for movements defined by confounder

CMDP : Confounding MDP

We created custom OpenAI gym envs which act as a probabilistic model for CMDP files, working on **circle.cmdp** to observe the effect of Confounding.

- Used a basic 3x3 grid environment with a binary confounder. The confounders used here are Clockwise (CW) and Counter Clockwise (CCW) direction enforcers.

- The agent starts in a random location and a binary confounder is uniformly sampled. The agent receives positive reward for moving alongside the border as per the confounder.

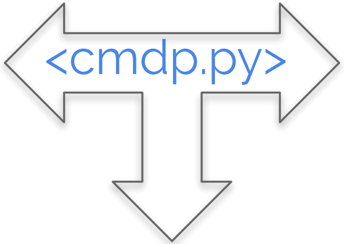
- We worked the algorithm for 10 timesteps and observed the effects of confounders on expected value conditioning on agents action vs expected value by intervening and making agent do a action.

PyroCMDP

OpenAI Gym environment which runs as a pyro probabilistic program.

```python
class PyroCMDP(gym.Env):  # pylint: disable=abstract-method
    metadata = {'render.modes': ['human']}

    def __init__(self, text, *, episodic, seed=None):
        self.model = parse(text)
        self.episodic = episodic
        self.seed(seed)

        if self.model.values == 'cost':
            raise ValueError('Unsupported `cost` values')

        self.discount = self.model.discount
        self.confounder_space = spaces.Discrete(len(self.model.confounders))
        self.state_space = spaces.Discrete(len(self.model.states))
        self.action_space = spaces.Discrete(len(self.model.actions))
        self.reward_range = self.model.R.min(), self.model.R.max()

        if self.model.U is None:
            self.U = (
                torch.ones(self.confounder_space.n) / self.confounder_space.n
            )
        else:
            self.U = torch.from_numpy(self.model.U.copy())

        if self.model.start is None:
            self.start = torch.ones(self.state_space.n) / self.state_space.n
        else:
            self.start = torch.from_numpy(self.model.start.copy())
        self.T = torch.from_numpy(self.model.T.transpose(1, 0, 2).copy())
        self.R = torch.from_numpy(self.model.R.transpose(0, 2, 1, 3).copy())

        self.D = None
        if episodic:
            # only if episodic
            self.D = torch.from_numpy(self.model.reset.T.copy())

        self.__time = None
        self.confounder = None
        self.state = None
        self.done = None
        self.action_prev = None
        self.reward_prev = None
```

File parser




```python
def render(  # pylint: disable=inconsistent-return-statements
    self, mode='human'
):
    if mode not in ('human', 'ansi'):
        raise ValueError('Only `human` and `ansi` modes are supported')

    # stream where to send the string representation of the env
    outfile = sys.stdout if mode == 'human' else io.StringIO()

    if self.action_prev is not None:
        ai = self.action_prev.item()
        print(f'action: {self.model.actions[ai]} (#{ai})', file=outfile)

    if self.reward_prev is not None:
        print(f'reward: {self.reward_prev.item()}', file=outfile)

    ui = self.confounder.item()
    print(f'confounder: {self.model.confounders[ui]} (#{ui})', file=outfile)

    si = self.state.item()
    print(f'state: {self.model.states[si]} (#{si})', file=outfile)

    if mode == 'ansi':
        with contextlib.closing(outfile):
            return outfile.getvalue()
```

<render()>

```python
def step(self, action):
    assert self.__time >= 0
    assert 0 <= self.state < self.state_space.n

    if not 0 <= action < self.action_space.n:
        raise ValueError(
            f'Action should be an integer in {{0, ..., {self.action_space.n}}}'
        )

    if self.done is None or self.__time is None:
        raise InternalStateError(
            'The environment must be reset before being used'
        )

    if self.done:
        raise InternalStateError(
            'The previous episode has ended and the environment must reset'
        )

    self.__time += 1

    state_next_dist = Categorical(self.T[self.state, action])
    state_next = pyro.sample(f'S_{self.__time}', state_next_dist)

    reward_dist = Delta(
        self.R[self.confounder, self.state, action, state_next]
    )
    reward = pyro.sample(f'R_{self.__time}', reward_dist)

    if self.episodic:
        done = self.D[self.state, action]
    else:
        done = torch.tensor(False)

    done_probs = torch.eye(2)[done.long()]
    done_dist = Categorical(done_probs)
    done = pyro.sample(f'D_{self.__time}', done_dist)

    info = {
        'T': self.T[self.state, action],
        'R': self.R[self.confounder, self.state, action],
    }

    self.state = state_next
    self.action_prev = action
    self.reward_prev = reward

    return state_next, reward, done, info
```

<step()>

Passing *circle.cmdp* file as an argument to <make_cmdp()>

```python
def main():
    env = make_cmdp(args.cmdp, episodic=True)
    env = TimeLimit(env, 10)

    agent_model_name = args.cmdp.split('/')[-1]
    agent_model = agent_models.get_agent_model(agent_model_name)

    values_df_index = 'E[G]', 'E[G | A=a]', 'E[G | do(A=a)]'
    values_df_columns = env.model.actions

    _, state = env.reset()
    for t in itt.count():
        print()
        print(f't: {t}')
        env.render()

        Qs_none = [
            infer_Q(env, action, 'none', agent_model=agent_model).item()
            for action in range(env.action_space.n)
        ]
        Qs_condition = [
            infer_Q(env, action, 'condition', agent_model=agent_model).item()
            for action in range(env.action_space.n)
        ]
        Qs_intervention = [
            infer_Q(env, action, 'intervention', agent_model=agent_model).item()
            for action in range(env.action_space.n)
        ]

        values_df = pd.DataFrame(
            [Qs_none, Qs_condition, Qs_intervention],
            values_df_index,
            values_df_columns,
        )
        print(values_df)

        action = torch.tensor(Qs_intervention).argmax()
        state, _, done, _ = env.step(action)

        if done:
            print()
            print(f'final state: {state}')
            print(f'Episode finished after {t+1} timesteps')
            break

    env.close()
```

< confounding_mdp.py >

```python
def infer_Q(env, action, infer_type='intervention', *, agent_model):
    if infer_type not in ('intervention', 'condition', 'none'):
        raise ValueError('Invalid inference type {infer_type}')

    if infer_type == 'intervention':
        model = pyro.do(trajectory_model, {'A_0': torch.tensor(action)})
    elif infer_type == 'condition':
        model = pyro.condition(trajectory_model, {'A_0': torch.tensor(action)})
    else:  # infer_type == 'none'
        model = trajectory_model

    posterior = Importance(model, num_samples=args.num_samples).run(
        env, agent_model=agent_model
    )
    return EmpiricalMarginal(posterior, 'G').mean
```

```python
def trajectory_model(env, *, agent_model):
    """trajectory_model

    A probabilistic program which simulates a trajectory by sampling random
    actions.  The sample return can be used to affect the trace likelihood such
    that the agent policy becomes

    $\\pi(action_0; state_0) \\propto \\exp(\\alpha return_0)$

    :param env: OpenAI Gym environment
    :param agent_model: agent's probabilistic program
    """
    env = deepcopy(env)

    # initializing the running return and discount factor
    return_, discount = 0.0, 1.0

    # with keep_state=True only the time-step used to name sites is being reset
    state, confounder = env.reset(keep_state=True)
    for t in itt.count():
        action = agent_model(f'A_{t}', env, (state, confounder))
        state, reward, done, _ = env.step(action)

        # updating the running return and discount factor
        return_ += discount * reward
        discount *= args.gamma

        if done:
            break

    pyro.sample('G', Delta(return_))

    return return_
```

Results

```
args: Namespace(cmdp='circle.cmdp', gamma=0.95, num_samples=1000)

t: 0
confounder: cw (#0)
state: s22 (#8)
                   up      down      left     right
E[G]           8.025261  8.025261  8.025261  8.025261
E[G | A=a]     8.025261  7.025261  8.025261  7.025261
E[G | do(A=a)] 7.522261  7.025261  7.520261  7.025261

t: 1
action: up (#0)
reward: 0.0
confounder: cw (#0)
state: s12 (#5)
                   up      down      left     right
E[G]           8.025261  8.025261  8.025261  8.025261
E[G | A=a]     8.025261  8.025261  6.075261  7.025261
E[G | do(A=a)] 7.543261  7.481261  6.075261  7.025261

t: 2
action: up (#0)
reward: 0.0
confounder: cw (#0)
state: s02 (#2)
                   up      down      left     right
E[G]           8.025261  8.025261  8.025261  8.025261
E[G | A=a]     7.025261  8.025261  8.025261  7.025261
E[G | do(A=a)] 7.025261  7.547261  7.504261  7.025261
```

• • •

```
t: 8
action: right (#3)
reward: 1.0
confounder: cw (#0)
state: s02 (#2)
                   up      down      left     right
E[G]           8.025261  8.025261  8.025261  8.025261
E[G | A=a]     7.025261  8.025261  8.025261  7.025261
E[G | do(A=a)] 7.025261  7.528261  7.528261  7.025261

t: 9
action: left (#2)
reward: 0.0
confounder: cw (#0)
state: s01 (#1)
                   up      down      left     right
E[G]           8.025261  8.025261  8.025261  8.025261
E[G | A=a]     7.025261  6.075261  8.025261  8.025261
E[G | do(A=a)] 7.025261  6.075261  7.534261  7.515261

final state: 0
Episode finished after 10 timesteps
```

Thank You