Tutorial

April 26, 2020

This is an overall tutorial of how to go about this project. The files can't actually be directly run in Jupyter Notebook since they require commandline arguments, but you can use the **sys** library and cell magic commands to execute the file while passing the commandline arguments as shown below. Be sure to go through the `requirements.txt` file to get all the libraries and repositories installed needed for this project to work.

1 Frozenlake implementations and Generalized MDPs

What we tried to achieve here:

1. 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.
2. 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.

The goal here was to solve the problem of implementing a related type of agent, the softmax agent which evaluates its own policy to compute its policy.

Walking through the code for related `contol_as_inference.py`. As mentioned before the code chunks cannot be run directly here but would require a command-line like call which is shown below where we demonstrate a sample output.

The code is distributed for 2 different implementations, one for FrozenLake environment and other for general MDPs, the breakdown would be shown below.

The *main*() function is used to provide choice of which implementation is to be run. It factors in the choice and generates the relative environment.

```
[ ]: 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()


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('mdp', help='`frozenlake` string or path to MDP file')
    parser.add_argument(
        '--policy',
        choices=['control-as-inference-like', 'softmax-like'],
        default='control-as-inference-like',
        help='Choose one of two control strategies',
    )
    parser.add_argument('--alpha', type=float, default=100.0) # likelihood␣
↪parameter
    parser.add_argument('--gamma', type=float, default=0.95) # discount factor
    parser.add_argument('--num-samples', type=int, default=2_000)
    args = parser.parse_args()

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

1.1 Implementation to solve frozen lake environment

We tried to create a softmax implementation FrozenLake but later realized that it was more similar to an implemtation of **control as inference**!

The following code is used to generate frozenlake environment trajectories. We use `pyro.factor()` is used to influence trace log-likelihood, it acts as soft-conditioning/filtering to select random trajectories which result in high sample return.

[ ]:
```
def trajectory_model_frozenlake(env, *, agent_model=None, factor_G=False):
    """trajectory_model_frozenlake

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

    :param env: OpenAI Gym FrozenLake 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
    for t in itt.count():
        action = agent_model(f'A_{t}', env, env.s)
        _, reward, done, _ = env.step(action.item())

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

        if done:
            break

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

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

    return return_
```

The $policy\_control\_as\_inference\_like()$ function is used toapply importance sampling to sample action site A_0 and display the marginal probabilities in tabulated format

```
[ ]: 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()
```

1.2 Implementation for General MDPs

This is a similar implementation of **control as inference** for the case of general MDP's. Before we were working on a predefined gym environment of Frozen lake but using the `make_mdp()` function in main, we make call to *PyroMDP* implementation which is done in `gym-pyro` repository. This generated and returns a probailistic environment which can be used to solve by out agent.

```
[ ]: 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_
```

This function works similar to $policy\_control\_as\_inference\_like()$ function but for the general MDP case.

```
[ ]: def softmax_like(env, *, trajectory_model, agent_model, log=False):
         """softmax_like

         :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
    """

    Qs = torch.as_tensor(
        [
            infer_Q(
                env,
                action,
                trajectory_model=trajectory_model,
                agent_model=agent_model,
                log=log,
            )
            for action in range(env.action_space.n)
        ]
    )
    action_logits = args.alpha * Qs
    action_dist = Categorical(logits=action_logits)

    if log:
        print('policy:')
        print(
            tabulate(
                [action_dist.probs.tolist()],
                headers=env.actions,
                tablefmt='fancy_grid',
            )
        )

    return action_dist.sample()
```

A sample output and demonstration of execution for the file `control_as_inference.py` .

[2]:
```
import sys
%run control_as_inference.py "frozenlake"
```

```
args: Namespace(alpha=100.0, gamma=0.95, mdp='frozenlake', num_samples=2000,
policy='control-as-inference-like')
---
t: 0
state:

SFFF
FHFH
FFFH
HFFG
policy:
    left     down     right     up

       0        1         0      0

reward: 0.0
---
t: 1
state:
  (Down)
SFFF
FHFH
FFFH
HFFG
policy:

    left     down     right       up

   0.005   0.9945         0   0.0005

reward: 0.0
---
t: 2
state:
  (Down)
SFFF
FHFH
FFFH
HFFG
policy:

    left     down     right     up

  0.0015        0    0.9985      0

reward: 0.0
---
t: 3
state:
  (Right)
SFFF
FHFH
FFFH
HFFG
policy:

    left     down     right     up
```

```
       0    0.565     0.435      0

reward: 0.0
---
t: 4
state:
  (Down)
SFFF
FHFH
FFFH
HFFG
policy:

    left     down     right     up

       0   0.0045    0.9955      0

reward: 0.0
---
t: 5
state:
  (Right)
SFFF
FHFH
FFFH
HFFG
policy:

    left     down     right     up

       0    0.002     0.998      0

reward: 1.0
final state:
  (Right)
SFFF
FHFH
FFFH
HFFG
Episode finished after 6 timesteps
```

From the output steps, we observe that out agent was able to solve optimally for the frozen lake environment in 6 timesteps to reach the goal.

2 Our final attempt at implementing Softmax

Previously we tried to implement softmax but didn't quite succeed. In this attempt we think we reached the closest to implementing a real and efficient softmax.

Here we have the implementation of 2 paths: 1. Frozen lake: Frozen lake environment based on pyro importance sampling 2. General MDP: We used `gridworld.mdp` to show implementation in general case where any environment can be defined in a `.mdp` format.

[ ]:
```
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()
```

The following function a probabilistic program for MDP environment trajectories using a presampled policy.

[ ]:
```
def trajectory_model(env, policy):
    """trajectory_model

    A probabilistic program for MDP environment trajectories using a presampled
    policy.

    :param env: OpenAI Gym FrozenLake environment
    :param policy: predetermined policy function
    """
    env = deepcopy(env)

    # running return and discount factor
    return_, discount = 0.0, 1.0
    for _ in itt.count():
        action = policy(env.state)
        _, reward, done, _ = env.step(action)

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

        if done:
            break

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

    return return_
```

The following model is used to performs inference to estimate $Q^\pi(s, a)$, then uses pyro.factor to modify the trace log-likelihood.

[4]:
```
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
```

We sample the policy using importance sampling on the entire above process. The action is chosen using the sample policy.

```
[ ]: 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]
```

We observed that it kinda works sometimes, but is very sensitive to hyper parameters. This was our final implementation of softmax, although we are not sure that its actually equivalent to the real softmax, a more formal proof is required for that which maybe a good idea for Future works on this.

Following line of code shows a sample exectuion of a file for the `gridworld.mdp`.

```
[4]: %run softmax_presample_policy.py gridworld.mdp
```

```
args: Namespace(alpha=5000.0, gamma=0.95, mdp='gridworld.mdp', num_samples=20)
---
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

       1        0       0       0

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

   north    south    east    west

       0        0       0       1

reward: -0.1
---
t: 3
state:
action: west
..+
. .-
…
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         0        1        0

reward: -0.1
---
t: 7
state:
action: east
…+
. .-
…
policy:

    north     south     east     west

     0.05       0.5     0.15      0.3

reward: 1.0
final state:
action: east
```

```
…+
.  .-
.…
Episode finished after 8 timesteps
```

We see that the agent was able to solve the problem in 8 timesteps and gain the final reward of 1.

3 Observing effect of Confounding on our agent's ability to solve the environment

The main goal here was to observe the effect of confounding by observing for conditioning vs intervening on action for a general RL problem using a Confounding MDPs file which can be thought as a special case of partially observable MDPs (POMDPs).

$E[R_t|S_t = s, do(A_t = a)] \neq E[R_t|S_t = s, A_t = a]$

We make use of the OpenAI gym environment framework and use a special format cmdp file. The cmdp file is derived from the encoding format for a MDP/POMDP problem, containing the states, rewards, confounders, actions and transition probabilities. The cmdp file is parsed into the environment using specially created parser from rl_parser repository which translates this file to a problem in gym environment which can be solved using traditional reinforcement learning techniques.

The code for this part is implemented in `confounding_mdp.py` file.

The following code is dependent on `PyroCMDP` implemetation from `gym-pyro` repository.

The following code is used to make a call to the Pyro file which creates samples and generates the confounding environment for the agent to solve.

```
from envs import make_cmdp
```

Here we are using circles.cmpd which is nothing but a 3 x 3 grid environment with a binary confounder. The counfounders here are Clockwise and Counterclockwise direction enforcers. The agent receives positive reward for moving alongside the border depending on the confounder.

The $main()$ function is where all the function calls are made. The Qs values are calculated by making recurring calls to the $infer\_Q()$ function and are displayed in a tabulated format.

We let the agent work for 10 timesteps and observed the effects of confounders on expected value conditioning on agents action vs expected value by making agent do A action. The agent model always tries to pick optimal action based on the intervention distribution. The confounding effect becomes more apparent since the agent model behaves differently according to it.

```
[ ]: 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()


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('cmdp', help='CMDP file')
    parser.add_argument(
        '--gamma', type=float, default=0.95, help='discount factor'
    )
    parser.add_argument(
        '--num-samples',
        type=int,
        default=1_000,
        help='number of samples to be used for importance sampling',
    )
    args = parser.parse_args()

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

The trajectory function is used to simulate a trajectory by sampling random actions. The parameters here as the Open AI gym environment and the agent's probabilistic program

```
[ ]: 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)) #agent model
             state, reward, done, _ = env.step(action) #environment model

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

             if done:
                 break

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

         return return_
```

The infer_Q function here is used to get posteriors for intervention and conditioning for every action A in working action space. We show the calculated effects in the tables displayed. G here stands for goal and A for action

[ ]:
```python
def infer_Q(env, action, infer_type='intervention', *, agent_model):
    """infer_Q

    Infer Q(state, action) via pyro's importance sampling, via conditioning or
    intervention.

    :param env: OpenAI Gym environment
    :param action: integer action
    :param infer_type: type of inference; none, condition, or intervention
    :param agent_model: agent's probabilistic program
    """
    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
```

A sample output and demonstration of execution for the file `confounding_mdp.py` .

[1]:
```
%run confounding_mdp.py circle.cmdp
```

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

t: 0
…
…
...
                      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  7.536261  7.025261

t: 1
action: left
…
…
```

```
...
                         up      down      left     right
E[G]              8.025261  8.025261  8.025261  8.025261
E[G | A=a]        6.075261  7.025261  8.025261  8.025261
E[G | do(A=a)]    6.075261  7.025261  7.527261  7.542261

t: 2
action: right
…
…
...
                         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)]    7.560261  7.025261  7.507261  7.025261

t: 3
action: up
…
...
…
                         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)]    7.513261  7.497261  6.075261  7.025261

t: 4
action: up
...
…
…
                         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  7.515261  7.505261  7.025261

t: 5
action: down
…
...
…
                         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)]    7.546261  7.506261  6.075261  7.025261

t: 6
action: up
```

```
...
…
…
                    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  7.502261  7.522261  7.025261

t: 7
action: left
...
…
…
                    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  7.518261  7.504261

t: 8
action: left
...
…
…
                    up      down      left     right
E[G]              8.025261  8.025261  8.025261  8.025261
E[G | A=a]        7.025261  8.025261  7.025261  8.025261
E[G | do(A=a)]    7.025261  7.494261  7.025261  7.504261

t: 9
action: right
...
…
…
                    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  7.533261  7.504261

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

E[G] here is the expectation of reaching goal when agent model is choosing the action by itself. The agent model always chooses the same optimal action.

E[G | A=a] here represents the expectation of reaching goal while observing for an action A=a in the action space.

E[G| do(A=a)] here is used to represent the expectation of reaching goal when we intervene and set the action A=a.

From the results we try to observe that conditioning and intervening give us two different values. To understand the results lets focus on say t: 8.

So here we observe that the agent just moved to left after the previous time step reaches state s00. Computing the expectations for the possible action space now, comparing the conditioning and do action on A=a we observe that the expectation for conditioning and intervening stays the same for up and left since those actions are not permitted as s00 represents the top left of the grid. Although when we see for the possible actions which are down and right it seems that conditioning overestimate the values for moving down and right whereas do operation shows that the values should be lower than what are being expected. This could show an observed effect of confounding which appears while conditioning. Moving to the time step 9 we see that the agent actually moved to right and the reward changed to 0.0 which shows that it was not the best action to take maybe it could have had a better reward if it moved down instead. At the end of 10 timesteps we see that the agent end at 0 as the final state.

Hence, we were able to explore the difference between “conditional” RL and causal RL and this concluded our presentation for the project.