A Causal Approach to Modeling Rational Agents

Fall 2019 Project Course Report

Christopher Botica
Student
Northeastern University
botica.c@husky.neu.edu

Dr. Robert Ness
Instructor
Northeastern University
r.ness@northeastern.edu

Dr. Olga Vitek
Instructor
Northeastern University
o.vitek@northeastern.edu

1 Introduction and Motivation

Recent advancements in Reinforcement Learning (RL) have caught the attention and produced excitement in the machine learning community and beyond. In many games, artificial agents surpass human players, such as Deepmind's AlphaGo in the board game "Go", and OpenAI Five in the team-based role-playing computer game "Dota 2". Starting with these examples, one can argue that current RL algorithms exceed human intelligence. However, even though state-of-the-art RL techniques obtain great results in terms of their cumulated reward, most do so without forming a conceptual understanding of the game or, at least, formalizing the causal concepts involved. In this sense, they form strong correlations that are advantageous to very specific game settings, but fall apart once these settings are slightly changed. On the other hand, human agents first form a causal model of their environment in terms cause and effect and reason about these models when deciding what actions to take.

Furthermore, recent computational advancements for inference suggest replacing existing inference methods with improved techniques. This can be accomplished under the Probabilistic Programming paradigm, which provides a powerful software environment for model-based machine learning and the support of random variables [1]. Under this paradigm, the generative model, conditioning on observed data, and the inference method are all separate: the generative model describes the forward generation process, while the inference method describes backwards reasoning to update prior beliefs.

2 Objectives

Our objectives for this semester have been to introduce causal modeling theory and code implementation for modeling rational agents. Specifically, we implement the agentmodels.org tutorial: 'Modeling Agents with Probabilistic Programs' [2] using a causal approach. The online book describes and implements agents as probabilistic programs for MDPs and POMDPs and uses a JavaScript-based probabilistic programming language, WebPPl, in their examples.

Introducing causal modeling theory and methods for RL agents hinges on our ability to model them in a probabilistic language. Due to its model-based approach, scalability, and access to deep learning tools in Python, Pyro, which is a probabilistic programing language in Python, is our preferred choice for this task.

Furthermore, we our objective is to extend the agentmodels.org implementation by using causal modeling theory, Pyro inference methods, and introduction of counterfactual regret to specific POMDPs [3].

By doing so in this course, we achieve the following:

- Causal model-based approach to modeling one-shot and sequential RL agents with clear separation between generative model and inference methods
- Novel introduction of causality theory in agent decision making, which is more robust against confounders and new environments
- Pyro implementation of causal agents (Integer line MDP)

We divide the remainder of the report as follows: in Section 2, we define our objectives. In Section 3, we distinguish between deliberate and reactive actions. In Section 4, we define our causal agents with pseudocode for implementation. In Section 5, we exemplify the use case of causal agents with a hypothetical bandit example. In Section 6, we present the challenges we faced during this semester. In Section 7, we present future work.

3 Deliberative Action Framework

In the general RL literature, actions admit two interpretations: reactive and deliberative. The former treats actions as ordinary events, whereas the latter treats actions as mechanism-modifying operations.

3.1 Reactive Actions

Reactive actions can be seen as consequences of agent's beliefs, state, and environment. They are interpreted as originating from outside of the agent and can be predicted by conditioning observational data. In this case, the agent will choose the action that maximizes the expectation of utility [4]

$$EU(x) = \sum_{y} P(y|x)u(y),$$

where EU represents the expected utility, $u(y)$ represents the utility of some outcome y, and x is an action. By using such a paradigm in sequential problems, the agent actions do not change the utility distribution at hand. Thus, it will use the evidence that was gathered to predict previous actions and the evidence that caused the act itself provides in order to choose whether to take that same action. Thus, this paradigm generates no information on the true, altering, aspect of the effect of an action on a system and falls apart in the presence of unobserved confounders. In essence, such actions yield observational distributions, and, in the same time, the agent attempts to answer interventional queries.

3.2 Deliberative Actions

On the other hand, deliberative actions are seen as choices in contemplated decision making. They are viewed from inside, and cannot be predicted from mere observational data. Furthermore, they do not provide evidence about the act itself, since, by definition, they are pending deliberation and transform into past acts once they are performed [4].

In this case, the agent seeks to maximize the expected utility [4]

$$EU(x) = \sum_{y} P\big(y\big|do(x)\big)u(y)$$

Such deliberate actions alter the 'world' the agent lives in. Whenever an agent intervenes upon a variable X (i.e. $do(x)$), it breaks the influence of the parents of X and sets it to the interventional value. Thus, such deliberate actions yield an interventional distribution, which can be easily used to answer higher-order queries such as effects of interventions or counterfactuals.

4 Causal Agents

We model our agents using the causal modeling tools and deliberative action framework. This allows the agent to form an understanding on the cause and effect of certain actions in the system. For example, to model the causal effect of taking an action a, on utility u, given a set of variables z, it finds the difference

$$P(u|do(action = a), z) - P(u|do(action = \neg a), z)$$

Furthermore, this can be extended to answer the counterfactual question: 'what would have happened to utility if action were a, given that we observed action was $\neg a$'. This question can be formalized also as counterfactual regret, which we discuss in Section 5.

To model our rational agents, we use action-environment framework depicted in Figure 1. To aid implementation, we structure our framework as Pyro pseudocode with and agent class, environment class, and individual methods.

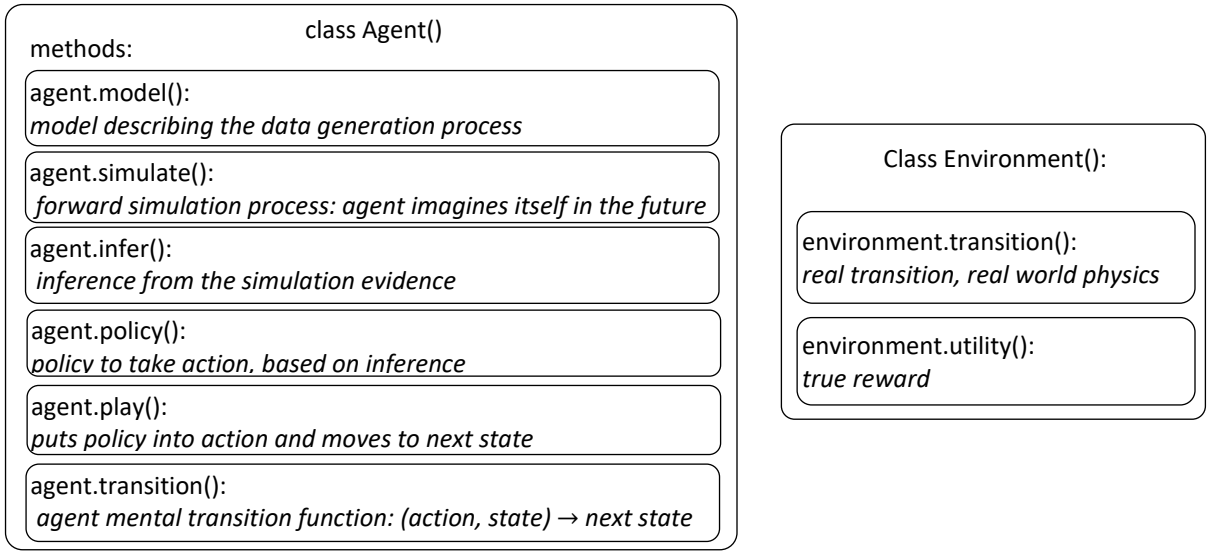


Figure 1: Action-Environment framework. The agent interacts with the environment by providing its state and action and the environment provides the next state and utility.

We define the transition function as $T: S \times A \to S$ mapping state-action pairs to state and the utility function $U: S \to \mathbb{R}$ maps the states to a real-valued utility.

4.1 One-Shot Decision Problems

In this case, the agent makes single choice between a set of actions, each of which has potentially distinct consequences. The structural causal model (SCM) for one-shot problems is illustrated in Figure 2.

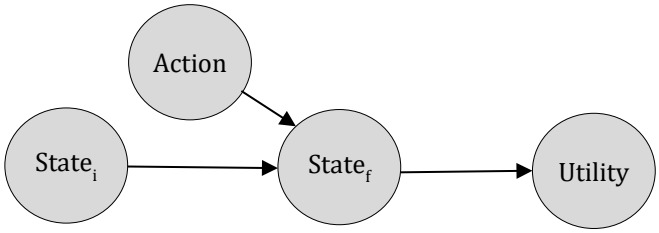


Figure 2: One-Shot Problem DAG

Based on agentmodels.org, we find three types of policies for this problem: the agent can maximize utility, maximize expectation of utility, of find a softmax of the expected utility.

In this case our environment for all three agents is:

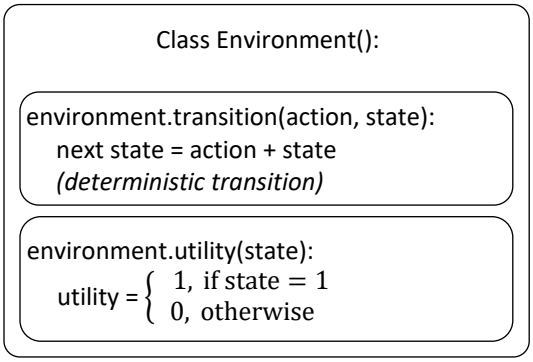


Figure 3: Utility-maximizing agent and environment classes for one-shot problems

4.1.1. Utility maximizing Agent

In this case, our policy is to take action a that maximizes utility:

$$choice = \underset{a \in A}{argmax}\, U\big(T(s,a)\big)$$

We construct the following utility-maximizing Agent, displayed in Figure 4.

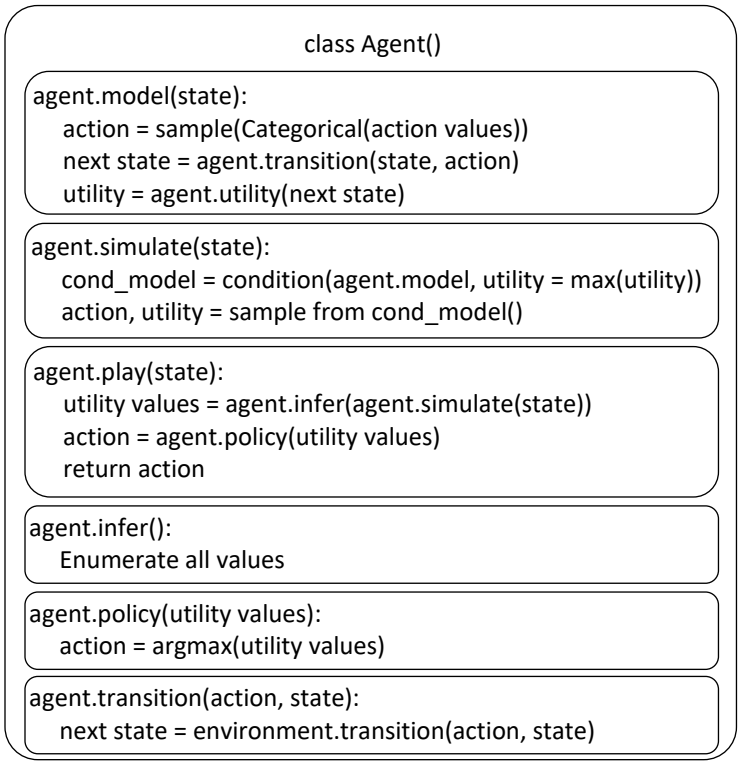


Figure 4: Utility-maximizing agent and environment classes for one-shot problems

Note that our set-up might seem unnecessary at this point. However, we keep the same agent model,

simulate, infer, and play format since it will become more useful for more complex agents.

4.1.2. Expected Utility - Maximizing Agent

In this case, our policy is to take action a that maximizes expected utility:

$$choice = \underset{a \in A}{argmax}\ \text{Expectation}\big[U\big(T(s,a)\big)\big]$$

We have the same environment and agent.transition as before. Our Agent is defined in Figure 5.

```
class Agent()

agent.model(state):
    action = sample(Categorical(action values))
    next state = agent.transition(state, action)
    utility = agent.utility(next state)

agent.simulate(state):
    action = sample from agent.model(state)
    interv_model = do(agent.model, 'action' = action)
    utility = sample from interv_model(state)

agent.play(state):
    utility posterior = agent.infer(agent.simulate(state))
    action = agent.policy(utility posterior)
    return action

agent.infer():
    Enumeratation / Importance Sampling

agent.policy(utility values):
    action = argmax(utility posterior)
```

Figure 5: Expected Utility-maximizing agent class for one-shot problems

In this case, the inference method calculates the expected utility from future simulations. This is particularly useful in situations in which we have noisy transitions and partially observed values.

4.1.3. Softmax Agent

In this case, our policy is to take action a that maximizes the softmax expected utility:

$$choice = \text{soft}\ \underset{a \in A}{argmax}\ \text{Expectation}\big[U\big(T(s,a)\big)\big], \text{ with factor } \alpha$$

In this case, the factor function expresses a soft-maximization condition. This is particularly useful in noisy situations in which a 'hard conditioning' is not desired. For example, in POMPDs it is infeasible to compute all scenarios to find a global maxima. Instead, it is better to have 'good enough' solutions. Specifically, the factor(x) function adds x to the unnormalized log-probability of the program execution of that variable. Then, agent.infer() uses the updated log-prob of each execution to determine the soft-max

We have the same environment as before. Our Agent is defined in Figure 6.

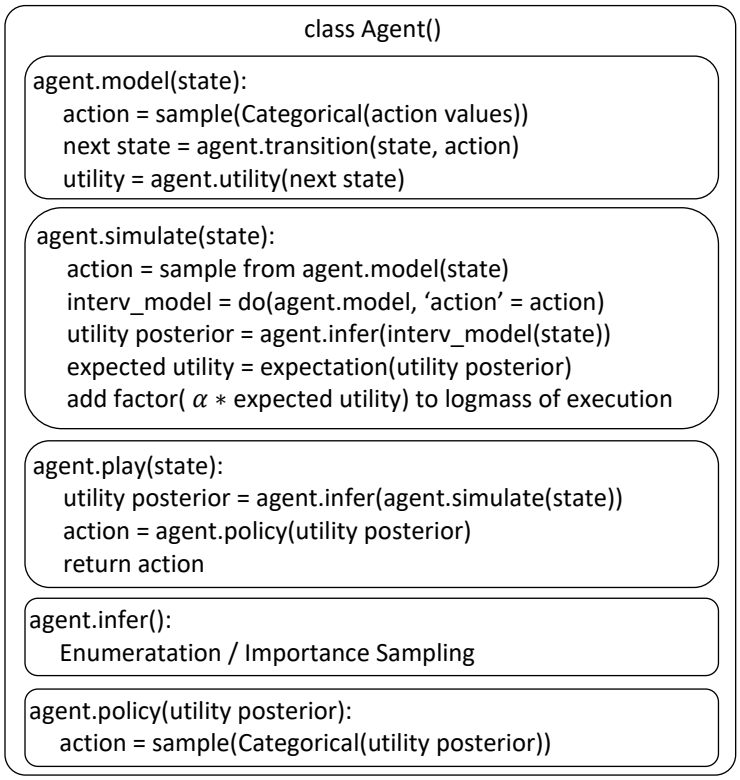


Figure 6: Expected Utility-maximizing agent class for one-shot problems

4.2 Sequential Decision Problems: MDPs

The agent's choice of actions now depends on the actions they will choose in the future. The rational agent coordinates with its future self. Based on agentmodel.org, we find the expected utility recursion: expected utility (EU) depends on both immediate utility and, recursively, on future expected utility:

$$EU[s] = U(s) + \text{Expectation}\big[U\big(T(s', a')\big)\big],$$

where s' is the next state after transition and a' is the action chosen at the next state. Throughout the remainder of this report, each agent policy will be softmax function of expected utility:

$$\text{choice}(\text{of action } a \text{ at state } s) \propto e^{\alpha * EU[s]}$$

4.2.1. MDP Agent

For each variant of the sequential MPD, we use the same agent and only change the environments. For each timestep until it reaches a terminal state (or, alternatively, the agent runs out of time), the agent will simulate itself in the future and infer the expected utility from its future self. We provide a detailed illustration of this process in Figure 7.

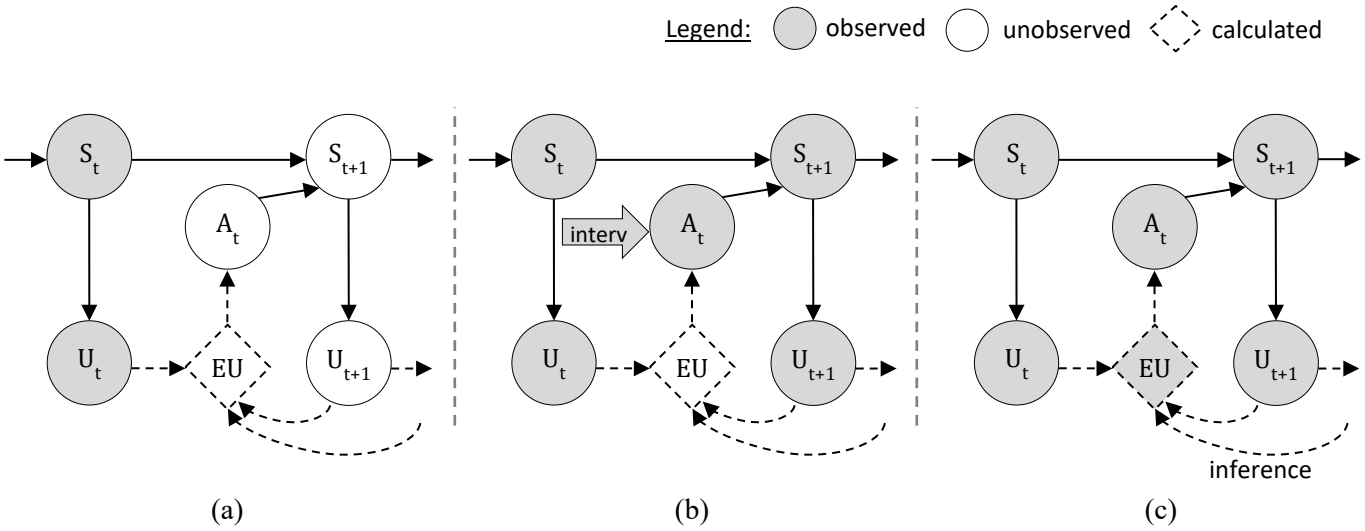


Figure 7: Agent SCM: simulation and inference at time step t. (a) Agent reaches time t with unknown future, (b) Agent simulates self in timestep $t+1$ by intervening on action variable (deliberative action) and continues for $t+2$, until it reaches the terminal state (not shown) (c) The simulated agent in timestep $t+1$ infers EU. The diamond shape illustrates that EU is not considered to be a random variable, part of the data generation process. Rather, it is an intermediate variable used to select the optimal policy.

Now we define our agent:

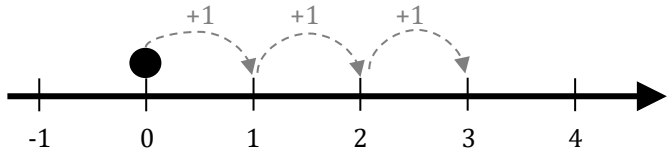
```
class Agent()

agent.model(state):
    action = sample(Categorical(action values))
    next state = agent.transition(state, action)
    utility = agent.utility(next state)

agent.simulate(state):
    while terminal state not reached:
        action = sample from agent.model(state)
        interv_model = do(agent.model, 'action' = action)
        next state, utility = sample from interv_model()
        future utility posterior = agent.infer(agent.simulate(next state))
        expected utility = utility + expectation(future utility posterior)
        add factor( α ∗ expected utility) to logmass of execution

agent.play(state):
    EU posterior = agent.infer(agent.simulate(state)
    action = agent.policy(EU_posterior)
    interv model = do(model, 'action':action)
    next state = sample from interv model(state)
    return action, agent.play(next state)

agent.infer():
    Enumeratation / Importance Sampling

agent.policy(utility posterior):
    action = sample(Categorical(utility posterior))
```

Figure 6: Expected Utility-maximizing agent class for sequential problems

4.2.1. MDP Environments

Because of our causal model-based approach, in which we separate the model and the inference, we can simply replace the inference and keep the model. In the same fashion, we separate our agent class from the environment class, such that we can keep the agent the same and just change the environment.

For our integer line MDP, the agent starts at position 0. The possible actions are -1 (go left), 0 (stay), and $+1$ (go right). The goal is to reach state $= 3$.



Class Environment():

environment.transition(action, state):
next state = action + state,
where action ∈ {−1, 0, +1}

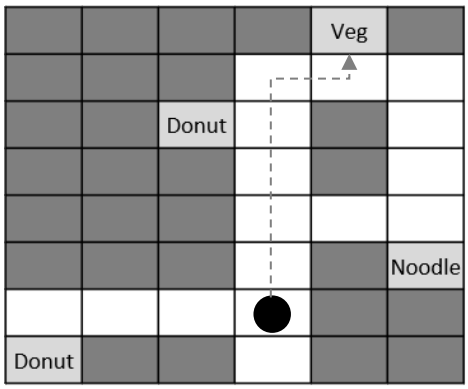
environment.utility(state):
utility = { 1, if state = 3
0, otherwise

(a) (b)

Figure 7: (a) Illustration of integer line MDP and example play (b) Integer Line MDP environment class

The full code and for the causal agent and integer line MDP can be found at our github repository: https://github.com/robertness/pyro_agents

In the agentmodels.org Griwdorld MDP, Agent Bob starts at position [3,1] and is looking for a place to eat. He has a complete view of the streets and restaurants nearby. His policy is to take a sequence of actions such that he eats at a restaurant he likes and he does not spend too much time walking. This is illustrated in Figure 8.



Class Environment():

environment.transition(action, state):
next state = action + state,
where action ∈ {up, down, left, right}

environment.utility(state):

$$utility = \begin{cases} 1, \textit{ if Donut} \\ 3, \textit{ if Vegetarian} \\ 2, \textit{ if Noodle} \\ -0.1, \textit{ otherwise (cost)} \end{cases}$$

(a) (b)

Figure 8: (a) Example of Gridworld MDP and example run (b) Gridworld environment

Note that variants of these MDPs include MDPs with noisy actions. For example, we can simply change the environment transition function to the following:

$$new\ state = \begin{cases} state + action, \textit{ with } p = 0.9 \\ state + alternate\ action, \textit{ with } p = 0.1 \end{cases}$$

4.3 Environments with hidden states: POMDPs

POMDP Definition: A tuple $\langle S,\ A(s),\ T(s,a),\ U(s,a),\ \Omega,\ O\rangle$ where S is the state space, A action space, T transition function, U utility function, Ω the finite space of observations the agent can receive, and $O{:}\ S \times A \to \Delta\Omega$ the observation function that maps the action and state to an observation $o \in \Omega$. Here, the agent tries to infer the belief from the observations. It does so by the following Bayesian update algorithm:

- Agent chooses an action a based on its belief distribution b, over its current state s
- Agent receives utility $u = U(s)$ at state s
- The Agent transitions to state $s' \sim T(s,a)$, where is gets observation $o \sim O(s')$ and updates its prior on the belief b' with observation o

According to agentmodels.org, we have the following POMDP Expected Utility of State Recursion:

$$EU[b] = U(s) + Expectation_{s',o,a'}(EU[b'\,]),$$

where $s' \sim T(s,a)$ and $o \sim O(s',a)$, b' is the updated belief function b on observation o, and a' is the softmax action the agent takes, given belief b'. However, the agent can't use this formulation to directly compute the best action, since the agent doesn't know the state. Instead we compute the expectation over possible states (note that direct enumeration is often intractable):

$$EU[b] = Expectation_{s\sim b}\left(U(s) + Expectation_{s',o,a'}(EU[b'\,])\right)$$

4.3.1. POMDP Agent

For each timestep until it reaches a terminal state, the agent will simulate itself in the future, use Bayesian update to infer its current belief, and infer the expected utility from its future self. This is illustrated in Figure 9.

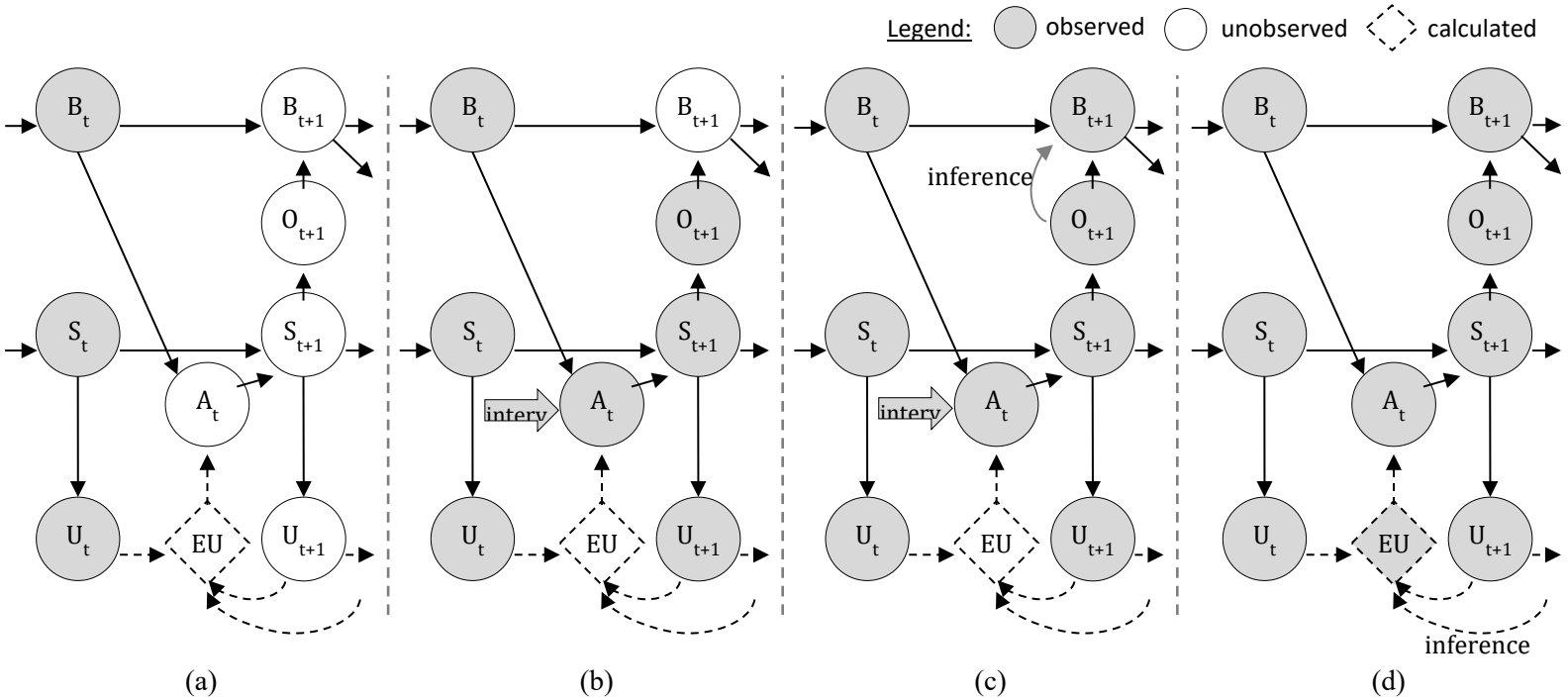


Figure 9: POMDP SCM: simulation and inference at time step t. (a) Agent reaches time t with unknown future, (b) simulates self in timestep $t+1$ by intervening on action variable and until it reaches the terminal state (not shown) (c) Agent in timestep $t+1$ performs the Bayesian update to infer its belief (d) The simulated agent in timestep $t+1$ infers EU, which it uses to compute the optimal policy

Now we define our agent in Figure 10.

```
class Agent()

agent.model(belief):
    state = sample(belief)
    action = sample(Categorical(action_vals))
    next_state = agent.mental_transition(state, action)
    utility_val = utility(next_state)
    observation = agent.observe(next_state)

agent.simulate(belief):
    while terminal state not reached:
        action, state = sample from model(belief)
        interv_model = do(agent.model, "action": action)
        cond_model = condition(interv_model, "state":state)
        utility_val = sample from cond_model(belief)
        updated_belief = update_belief(cond_model, belief)
        next_state, next_utility_val = sample from cond model(updated belief)
        future_utility_posterior = agent.infer(agent.simulate(updated_belief))
        current_EU = utility_val + expectation(future_utility_posterior)
        add factor(α ∗ EU) to logmass of execution

agent.observe(state):
    return environment.observe(state)  (perfect observation)

agent.play(belief):
    EU_posterior = agent.infer(agent.simulate(belief))
    action = agent.policy(EU_posterior)
    interv_model = do(model, "action"=action)
    updated_belief = update_belief(interv_model, belief)
    return action, agent.play(updated_belief)

def updated_belief(model, belief):
    observation = sample from model(belief)
    cond model = cond(model, 'observation' = observation)
    belief posterior = agent.infer(cond model)
    return belief posterior

agent.infer():
    Enumeratation / Importance Sampling

agent.policy(utility posterior):
    action = sample(Categorical(utility posterior))
```

Figure 10: POMPD Agent

And the POMDP Environment in Figure 11.

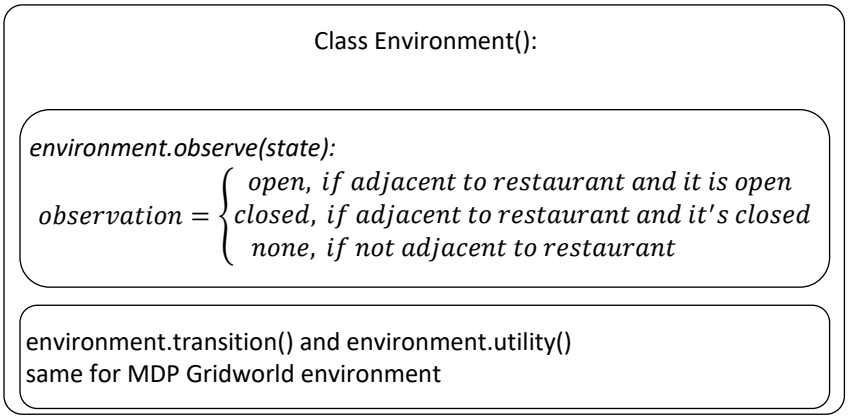


Figure 11: POMDP Environment

5 Example Use-Case: Counterfactual Regret

Due to our causal-modeling approach, we can employ our agents in different situation and just change the environment and inference method. This methodology is especially useful in the case of unobserved confounders. Let's first consider the simple multi-armed bandit problem, a variant of the POMDP:

5.1 Standard Bandit

The standard Multi-armed Bandit is a simplified variant of the POMDP: it has only 1 state and multiple actions (number of arms). Figure 12 illustrates the simple SCM for the bandit.

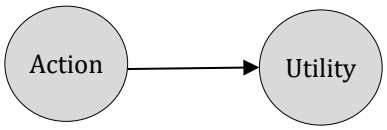


Figure 12: Simple Bandit DAG

The agent is certain about its belief of state, but uncertain about belief of hidden probability distribution mapping arms to prizes. In comparison, the sequential POMDP agent infers the distribution of EU at each step, whereas the Bandit agent infers the payout probability distribution for only one step. In this bandit setting, there were two machines, each with its own hidden prior payout probability distribution, and the agent's goal was to maximize the reward.

To show the use case of our causal approach, we first employ a standard (non-causal) sampling algorithm called Thompson Sampling. We can model this using our approach, illustrated in Figure 13.

class Agent()

agent.model():
arm = sample(Categorical(arm_vals)
utility = environment.utility(arm)

agent.simulate(belief):
arm = sample from model()
$P(arm = optimal) = \frac{belief_{arm}[wins]}{belief_{arm}[wins] + belief_{arm}[losses]}$

agent.play(belief):
arm = agent.policy(belief)
interv_model = do(model, "arm": arm)
utility = sample from interv_model()
update belief (#wins and # losses for arm)
arm_posterior = infer(agent.simulate(belief))
regret = max(hidden_prob_distr) – max(arm_posterior)

agent.infer():
Enumerate all values

agent.policy(belief):
(Standard Thompson Sampling)
choice = argmax_{arm}(sample(Beta($\text{belief}_{wins,\ arm}$, $\text{belief}_{losses,\ arm}$)))

Class Environment():

environment.utility(arm):
$hidden\ probs = \begin{cases} arm_1\ has\ payout\ p_1 \\ arm_2\ has\ payout\ p_2 \end{cases}$
utility = 1 with prob p_{arm}, 0 with $1 - p_{arm}$

Figure 13: Standard Bandit Agent and Environment.

As shown in Bareinboim et al., the regular bandit fails in the case in which confounders are present [3]. This result is illustrated in Figure 14.

5.2 Causal Bandit

Now, consider the following situation: in which we have an unobserved confounder, Z. For example, this can be the influence a casino has on flashing machines for drunk agents. Now, the updated DAG is shown in Figure 12.

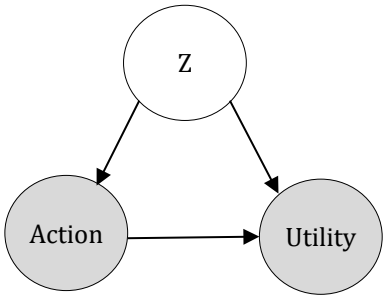


Figure 12: Simple Bandit DAG

In the presence of unobserved confounders, we find that traditional bandit algorithms, that rely on the

classical definition of reward as the arm with the highest associated payout, are unable to correctly learn the environment and are actually no different than random guessing [3]. Instead of taking actions purely based on maximizing some reward function, human agents reason about their decisions using *counterfactual regret*. That is, "had I made a different decision, things would have turned out better." Thus, by taking actions that avoid regretful outcomes, based on counterfactual reasoning of past consequences, human agents make use of powerful cognitive machinery – i.e., are truly "intelligent."

Formally, this is defined as

$$P(Y_{X=x} | X = x) - P(Y_{X=x'} | X = x).$$

This reads: given that $X = x$ and $Y = y$ were observed, what would have the effect of $X = x'$ been on Y? Note that the lower-case notation $Y_{X=x}$ is equivalent to $Y|do(X = x)$. Based on the summer semester, we frame the multi-armed bandit as a causal model and introduced counterfactual regret. Our code and results from the summer semester project can be found at our GitHub repo: https://github.com/tylernickr/causalbandits.

In this new framework, we show that the bandit is able to learn its environment and differentiate between the observational and interventional distributions under confounding. We call this Causal Thompson Sampling. Unlike the Standard Thompson Sampling Agent, the Causal Thompson Sampling Agent makes the confounder explicit in its structured causal model (SCM). According to the DAG defined in Figure 12, we define our SCM as follows:

- The confounder, Z, is sampled from a hidden distribution:

$$Z \sim Hidden\ Distribution$$

- The agent action, A, is influenced by the confounder. We call this function $intuition$, which can be stochastic or deterministic:

$$A = intuition(Z)$$

- And finally, utility, U, is depends both on Z and A and can be modeled using the agent.utility method:

$$U = \text{agent.utility}(Z, A)$$

The Standard Thompson Sampling Agent only sees the observational data: $P(U|A)$. Instead, we calculate the Effect of the Treatment of the Treated (ETT):

$$ETT = P(U_{A=a} | A = a) - P(U_{A=a'} | A = a),$$

where U and A are the binary utility and action variables, respectively, a is the agent intuition of best action found by $a = intuition(z)$ and a' is the agent counterintuition (for the binary case).

Using our SCM and the rules of do-calculus, we derive a closed form solution to the ETT.

First, from [4] we have that conditioning on the value of the counterfactual is equivalent to just conditioning. Thus,

$$P(U_{A=a} | A = a) = P(U|A = a)$$

Next, we use the product rule and marginalize over our confounder:

$$P(U_{A=a'} | A = a) = \sum_z P(U_{A=a'}, Z = z | A = a)$$

$$= \sum_z P(U_{A=a'} | Z = z, A = a) P(Z = z | A = a)$$

Since Z blocks the backdoor path between A and U, by [4], we have that the variable A is conditionally independent of $U_{A=a'}$, given Z. Therefore, we can replace $A = a$ with $A = a'$ in the first term of the product such that

$$P(U_{A=a'} | A = a) = \sum_z P(U_{A=a'} | Z = z, A = a) P(Z = z | A = a)$$

$$= \sum_z P(U | Z = z, A = a) P(Z = z | A = a),$$

Thus, the ETT becomes

$$ETT = \sum_z P(U | Z = z, A = a) P(Z = z | A = a) - P(U | A = a)$$

Now, we define our algorithm for the causal bandit (i.e. Causal Thompson Sampling), based on [3].

For the purpose of notation, let Q_1 be

$$Q_1 = P(U_{A=a'} | A = a)$$

$$= \sum_z P(U | Z = z, A = a) P(Z = z | A = a),$$

as we found before.

Similarly, let Q_2 be

$$Q_2 = P(U_{A=a} | A = a)$$

$$= P(U | A = a)$$

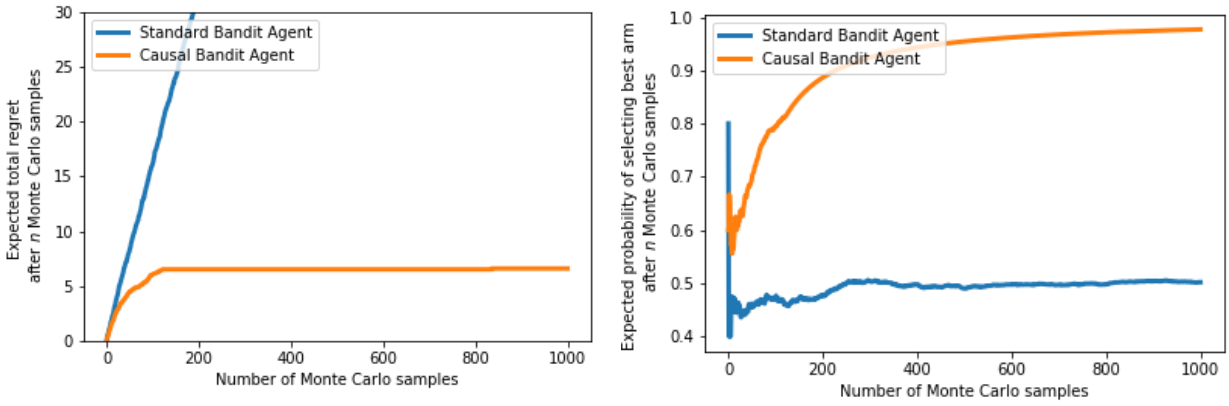
Similar to the Standard Thompson algorithm, we choose the argument that maximizes the updated Beta distribution and we use posterior predictive of the Beta-Bernoulli distribution to model payout probabilities. However, based on Bareinboim et al., we bias the arguments of the distribution using the ETT as follows:

- Get intuition for trial: $a =$ agent.policy(current belief)
- Initialize the weights for each arm: $w = [1,1]$
- Compute the weighting strength: $bias = 1 - |Q_1 - Q_2|$
- Choose the action to bias: if $Q_1 > Q_2$, then $w_a = bias$, else $w_{a'} = bias$

Finally, the agent chooses the arm that minimizes counterfactual regret. Let α_i be the number of successes when choosing the agent's intuition on machine i and β_i the number of failures. Thus, the choice is

$$choice = \underset{i}{argmax}\big(sample\big(Beta(1 + \alpha_i, 1 + \beta_i)\big) * w_i\big)$$

We implement these changes and add counterfactual regret to our standard bandit agent. Now, for the hypothetical case presented in [3], the causal bandit agent is able to learn the hidden probabilities even in the presence of a confounder. We illustrate this in Figure 14.



(a) (b)

Figure 14: (a) Expected total regret of our multi-armed bandit strategies (b) Expected probability of selecting best arm for each bandit strategy. In both cases the standard bandit agent is unable to converge, whereas the causal bandit converges to the optimal solution after just a couple hundred Monte Carlo samples

6 Challenges

Due to the novelty of this problem, we encountered many unique challenges in both theory and implementation.

6.1 Theory

Separating the model and inference methods for sequential agents proved to be difficult due to the recursive nature of the solution. Since the agent infers on simulated futures, it is only natural to assume that whatever generative model the agent may have also references such inference. This is the approach taken by agentmodels.org and most other sequential RL modeling. However, the ability to separate the agent generative model and its inference method proves to be much more useful, allowing for additional queries of the model (such as interventional or counterfactual queries) and the ability to easily replace the inference method.

Identifiability for effect of actions on utility when action is dependent on past/future variables ……

6.2 Implementation

The key aspect to this semester's research was the implementation. This is particularly important, since it lays the groundwork for any future extensions, but equally difficult. The main implementation challenges we solved are:

- Sequential sampling in Pyro. The Pyro 'sample' operation requires that the state name of the sampled variable to be unique. Thus, Pyro is not optimized for sequential decision making since it will produce an error whenever we sampling future variables with the same names. To find a workaround we renamed each variable according to the depth of the recursion and additional variables. For example, `utility` becomes `utility at state $s$ and time left $t$`.

- Slow inference and memoization. Inferring utility in Pyro using Importance Sampling was especially slow due to the exponential number of samples. Regularly, efficient programs make use of repetition in the program executions and are able to automatically store certain variables using memoization. Due to the sequential aspect of the problem, MDP and POMDP agents can take multiple paths to reach the same state. Inferring the utility in that state for each trajectory is a waste of computational resources and can be reduced by using memoization. However, Pyro does not work easily with memoization. To overcome this, we built a custom memoization function that saves each parameter using Pyro.param.

- Pyro and WebPPL incompatibility. Many built-in functions, such as infer() and factor() in WebPPL do not exist in Pyro or are incompatible. Therefore, in order to follow the agentmodels methods, we had to build our own functions based on help from the Pyro developers.

- Custom WebPPL environments. Besides the integer-line MDP, the environments presented in the agentmodels.org tutorial are custom built using WebPPL, and are not compatible with Python, or any open source Python environments such as OpenAI Gym. Therefore, the time spent trying to convert the WebPPL Gridworld environments to Python was not beneficial to us.

7 Path Forward

By building these techniques and implementation, we lay the groundwork for future research in modeling causal agents. Using our structural causal model-based approach, students or industry practitioners can easily extend our work to include more complex situations by simply replacing the inference methods and environment classes.

For any such future research, it is important to take note of our current realizations and impediments described in Section 6. We recommend that any future extensions of this work diverge from closely following the agentmodels.org environments. Instead, we recommend custom Python environments for MDPs and POMDPs or, better yet, the use open source environments from OpenAI Gym.

Additionally, we recommend the future researcher to include custom confounders in our POMDP example, in order to showcase the robustness of causal agents vs classical RL agents.

Furthermore, we recommend continuing the parallel to the agentmodels.org tutorial in modeling POMDPs with short horizon, cognitive biases, and learning from other agents.

The full code and for the causal agent and integer line MDP can be found at our github repository: https://github.com/robertness/pyro_agents

References

[1] Model-based machine learning: https://royalsocietypublishing.org/doi/full/10.1098/rsta.2012.0222#d3e1598

[2] Modeling Agents with Probabilistic Programs https://agentmodels.org/

[3] Bandits with Unobserved Confounders: https://ftp.cs.ucla.edu/pub/stat_ser/r460.pdf

[4] Pearl, Judea, *Causality*, 2nd edition, Cambridge University Press, 2009