Pearl – Production-ready Reinforcement Learning AI Agent Library

Presenters: Zheqing (Bill) Zhu, Jalaj Bhandari, Yi Wan
Contributed by: Rodrigo de Salvo Braz, Daniel Jiang, Yonathan Efroni, Liyuan Wang, Ruiyang Xu, Hongbo Guo, Alex Nikulkov, Dmytro Korenkevych, Urun Dogan, Frank Cheng, Zheng Wu, Wanqiao Xu

Applied Reinforcement Learning Team
AI at Meta

Meta

Agenda

Meta

01 | Who We Are

Applied Reinforcement Learning team @ AI at Meta

- We own the central reinforcement learning platform that supports dozens of applications across Meta

Applied Reinforcement Learning team @ AI at Meta

- We own the central reinforcement learning platform that supports dozens of applications across Meta

- We conduct state-of-the-art research that helps bridge the gap between current reinforcement learning algorithms and real-world impact

Applied Reinforcement Learning team @ AI at Meta

- We own the central reinforcement learning platform that supports dozens of applications across Meta
- We conduct state-of-the-art research that helps bridge the gap between current reinforcement learning algorithms and real-world impact
- We are also hands-on developing reinforcement learning agents that will benefit people and advertisers using Meta

02 | Introducing Pearl

Meta

PEARL'S MISSION

Enable production-ready reinforcement learning AI agents that adapt to a diverse set of real-world challenges.

MOTIVATION

- Sequential decision making is prevalent in real-world applications
- Examples: generative AI, recommender systems, robotics
- Calls for RL-based AI agents that can adapt to various real-world applications

HIGHLIGHTS

- A **diverse set of reinforcement learning features** that are not commonly covered by reinforcement learning libraries

HIGHLIGHTS

- A **diverse set of reinforcement learning features** that are not commonly covered by reinforcement learning libraries
- An AI agent with **modular design** that can **mix-and-match** multiple reinforcement learning features to address varieties of problems in real-life

HIGHLIGHTS

- A **diverse set of reinforcement learning features** that are not commonly covered by reinforcement learning libraries
- An AI agent with **modular design** that can **mix-and-match** multiple reinforcement learning features to address varieties of problems in real-life
- Pytorch-native and easy to integrate with production systems

QR Code for Repo

03 | Sequential Decision Making in Real Life

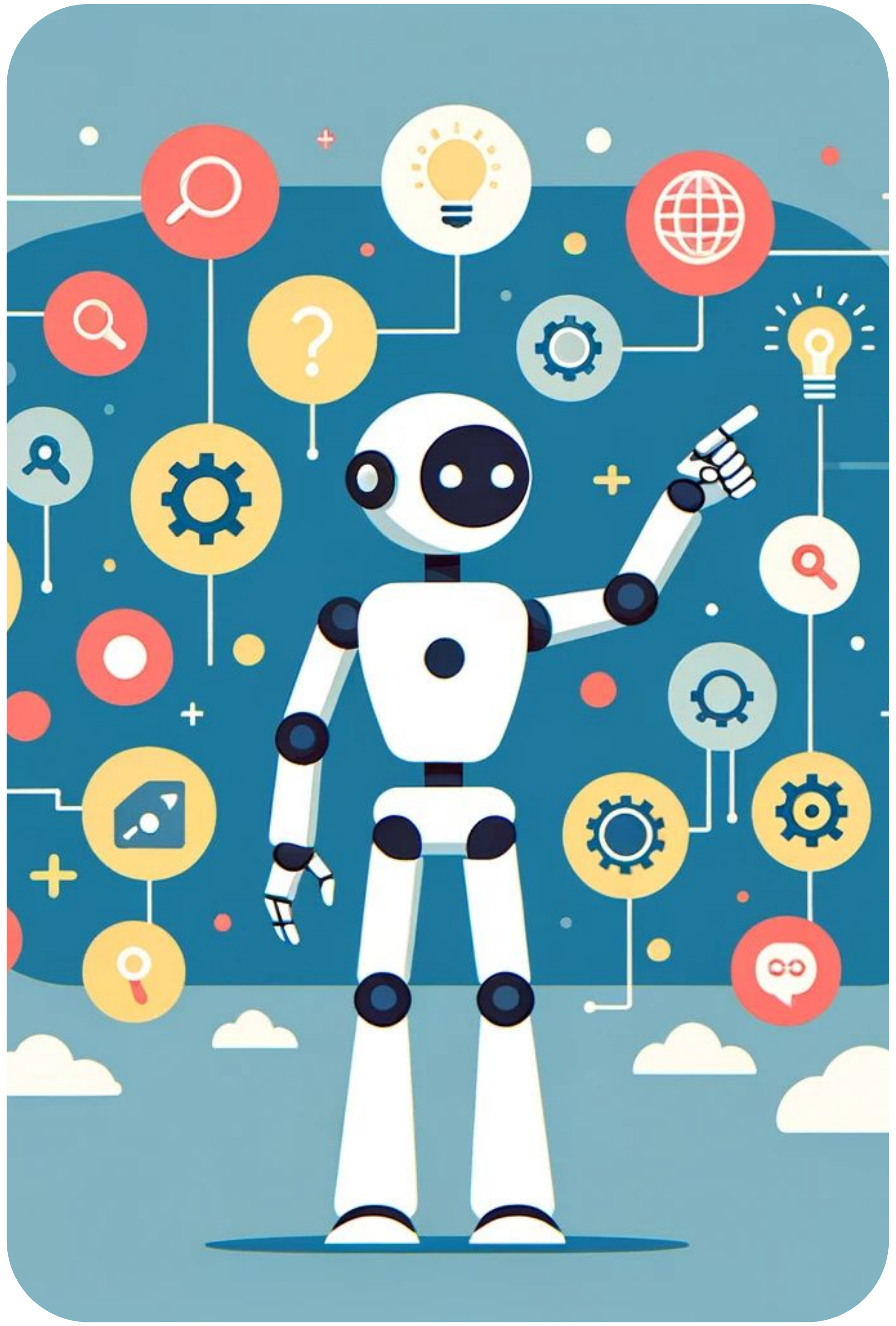
Meta

Recommender Systems

A sequence of recommendations that drives people’s long-term satisfaction

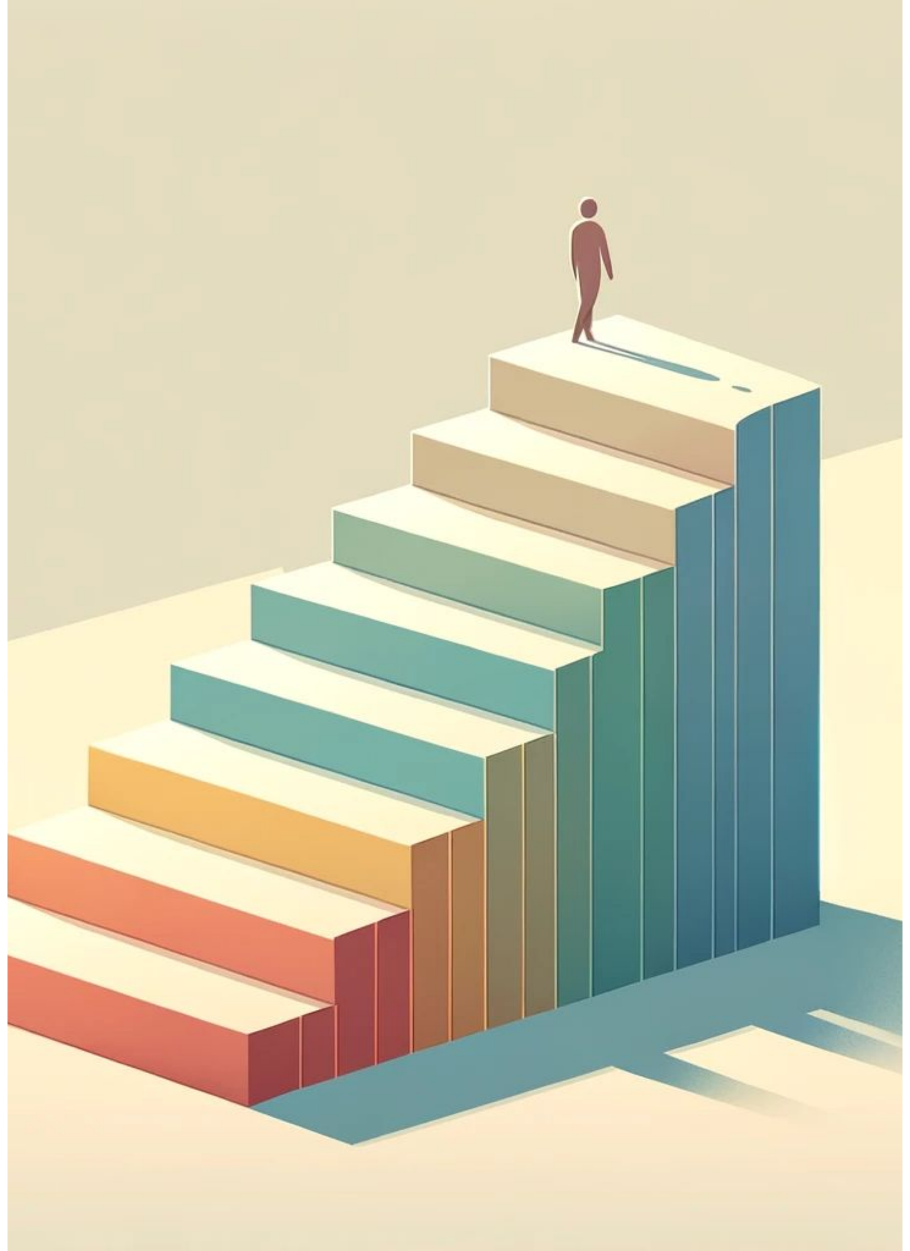
Creative Selection

Explore the best creative that can bring maximal customer traction



Auction Pacing

Deliver maximal advertising campaign outcome by sequentially decide how much budget to allocate in the next minute



Robotics

Making decisions on how to complete tasks in a unknown domain safely with only part of world observable

Supply Chain

Coordinate logistics across hundreds of sites to maximize throughput and minimize delay

04 | An Intro to Reinforcement Learning

Agent - Environment Interface

- Agent executes an action and receives an observation and a reward from the environment

Agent - Environment Interface

- Agent executes an action and receives an observation and a reward from the environment

- The observation might only contain limited information and reward might be sparse

Agent - Environment Interface

- Agent executes an action and receives an observation and a reward from the environment

- The observation might only contain limited information and reward might be sparse

- Some actions might also be dangerous

Agent - Environment Interface

- Agent executes an action and receives an observation and a reward from the environment
- The observation might only contain limited information and reward might be sparse
- Some actions might also be dangerous
- Available actions might change over time

Optimization Target

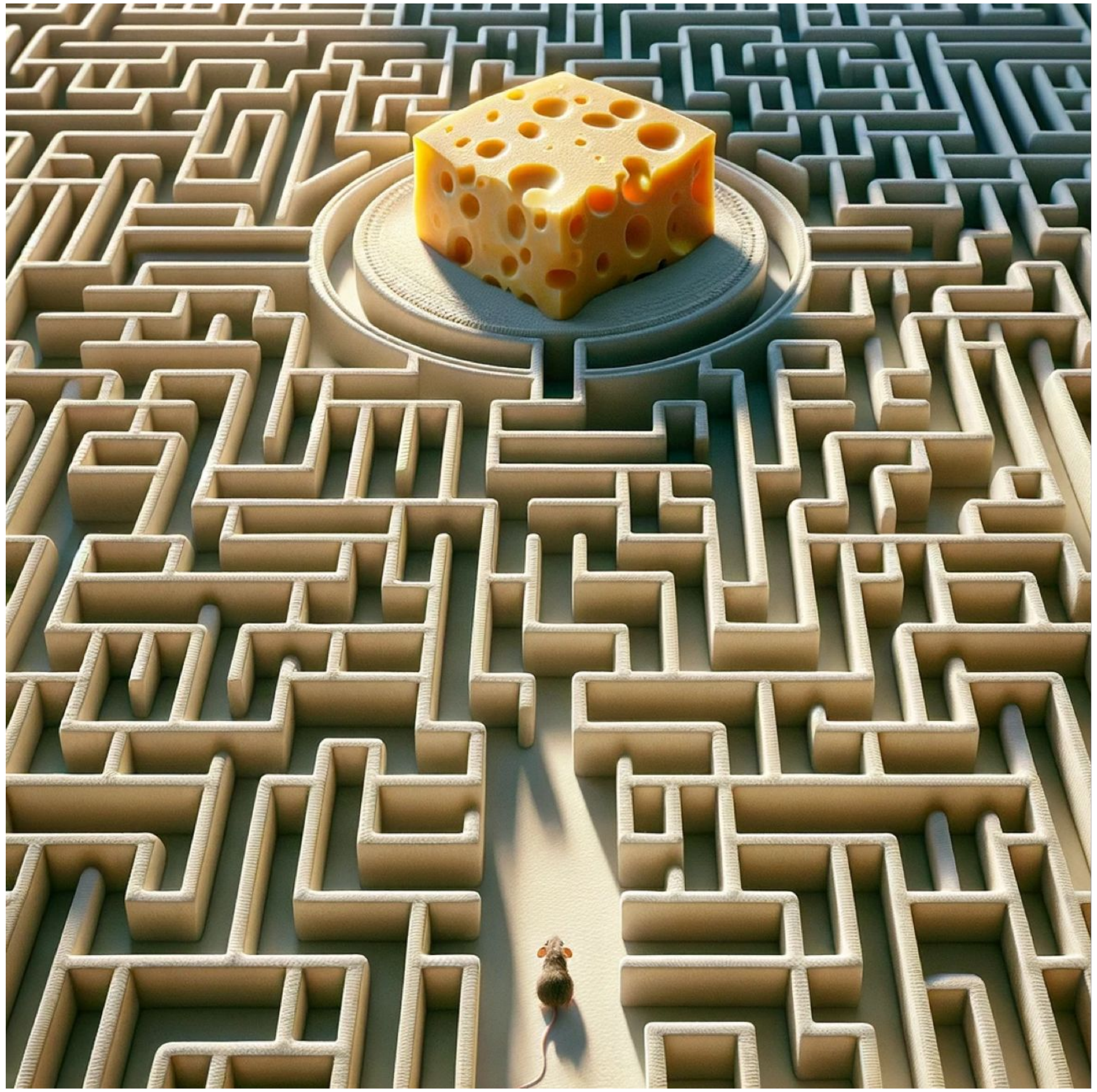
Cumulative reward throughout the lifetime of the agent

Policy Learning

Find a strategy that maximizes the cumulative reward

Sparse Reward

Reward might only surface after a long sequence of interactions and not often awarded.

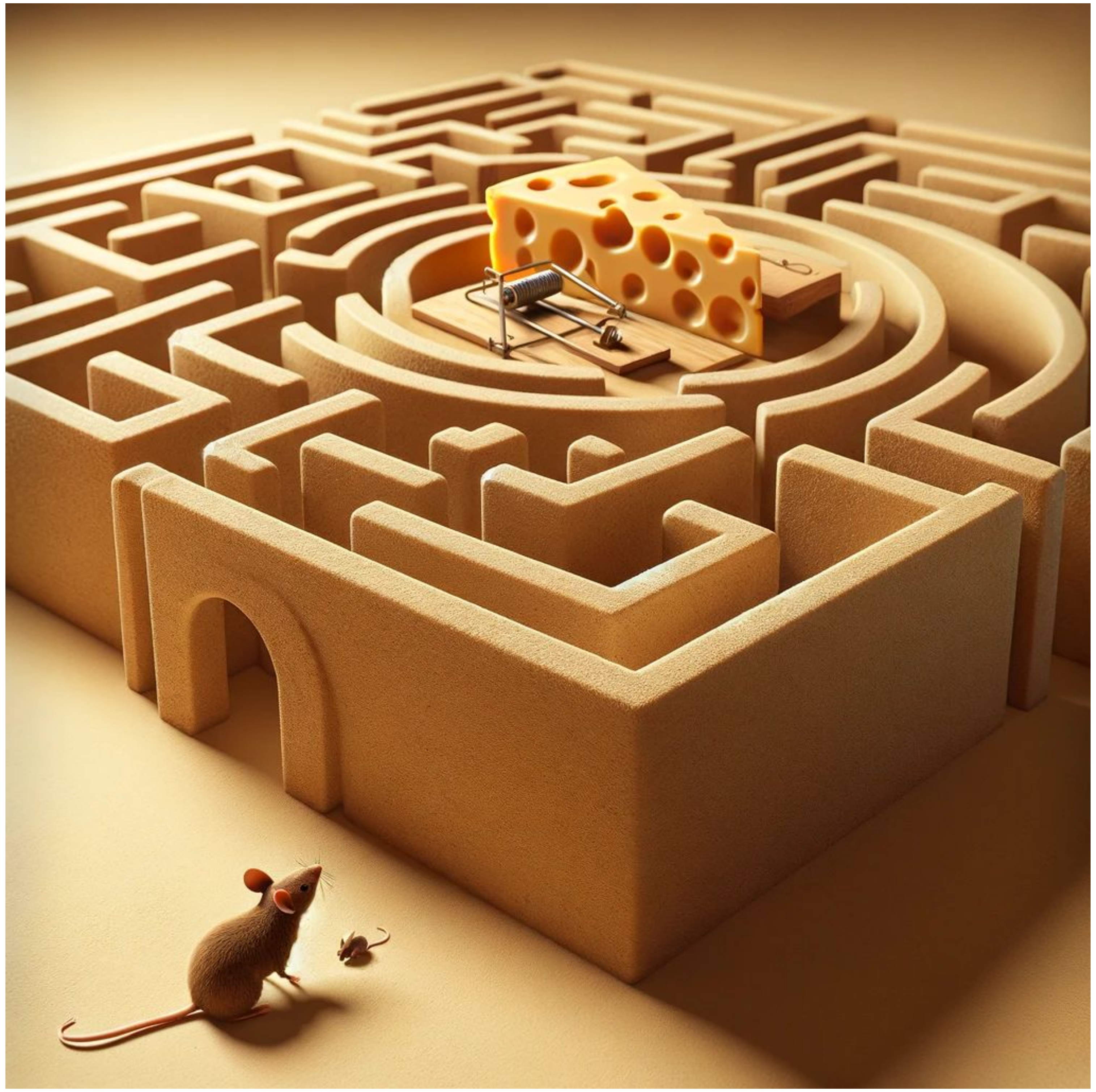


Partial Observability

Agent might not always be able to see where they are and it has to identify where it is from the past interactions.

Dangerous States and Actions

There might be states and actions that might result to catastrophic consequences.

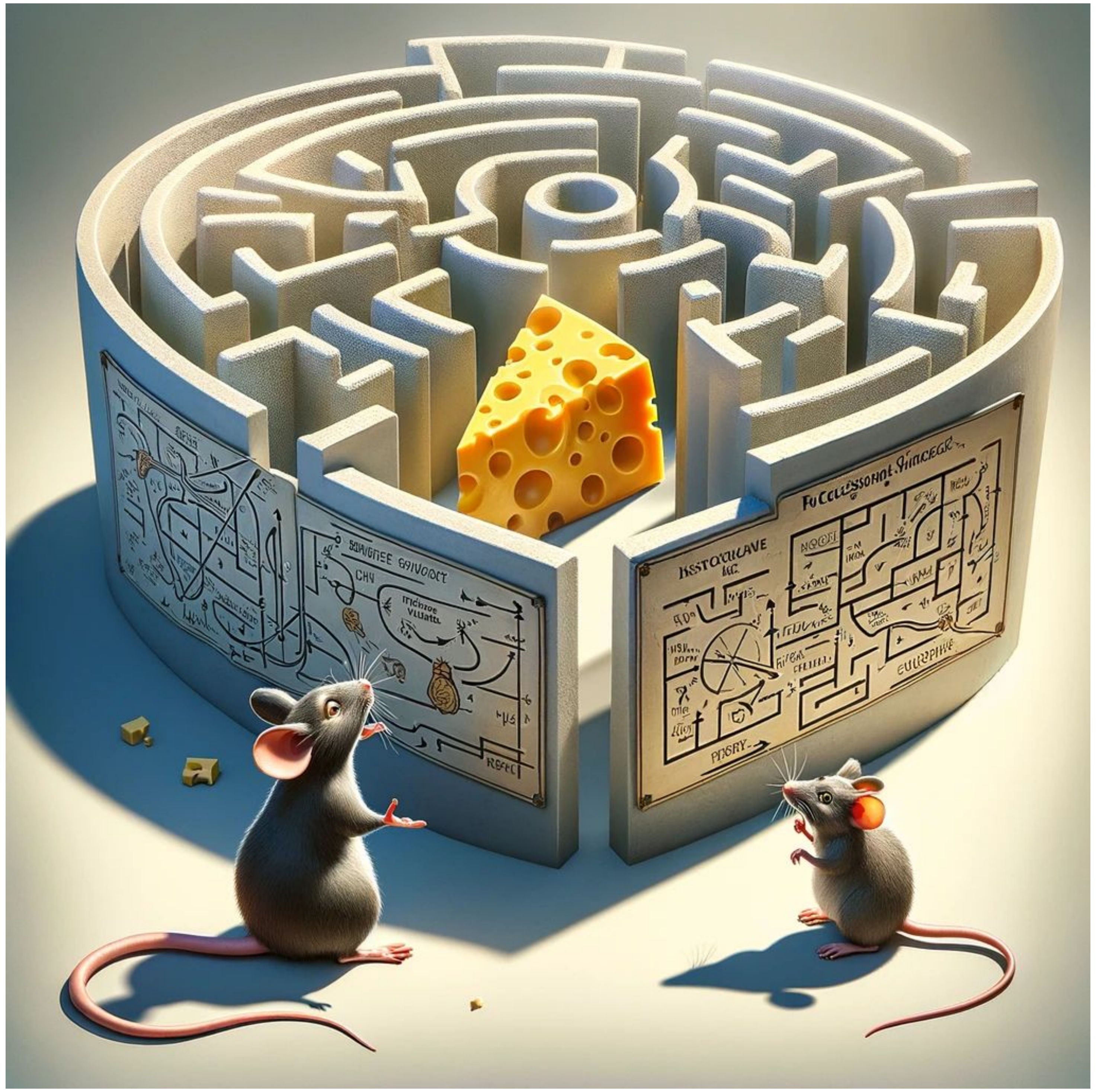


Risky States and Actions

There might be states and actions that might have high risk of leading into bad outcomes.

Offline Knowledge

Agent might be able to retrieve offline knowledge without interacting with the target environment.



Dynamic Available Actions

Agents can be offered a different set of available actions within a single environment.

05 | Why Pearl Stands Out

Coming Back to Our Real-life Examples

Challenges		Recommender Systems				
Sparse Reward		✓				
Dangerous and Risky State and Actions						
Partial Observability		✓				
Changing Action Space		✓				
Offline Learning		✓				

Coming Back to Our Real-life Examples

Challenges		Recommender Systems	Auction bidding			
Sparse Reward		✓	✓			
Dangerous and Risky State and Actions			✓			
Partial Observability		✓	✓			
Changing Action Space		✓				
Offline Learning		✓	✓			

Coming Back to Our Real-life Examples

Challenges		Recommender Systems	Auction bidding	Creative Selection		
Sparse Reward		✓	✓	✓		
Dangerous and Risky State and Actions			✓			
Partial Observability		✓	✓			
Changing Action Space		✓		✓		
Offline Learning		✓	✓	✓		

Coming Back to Our Real-life Examples

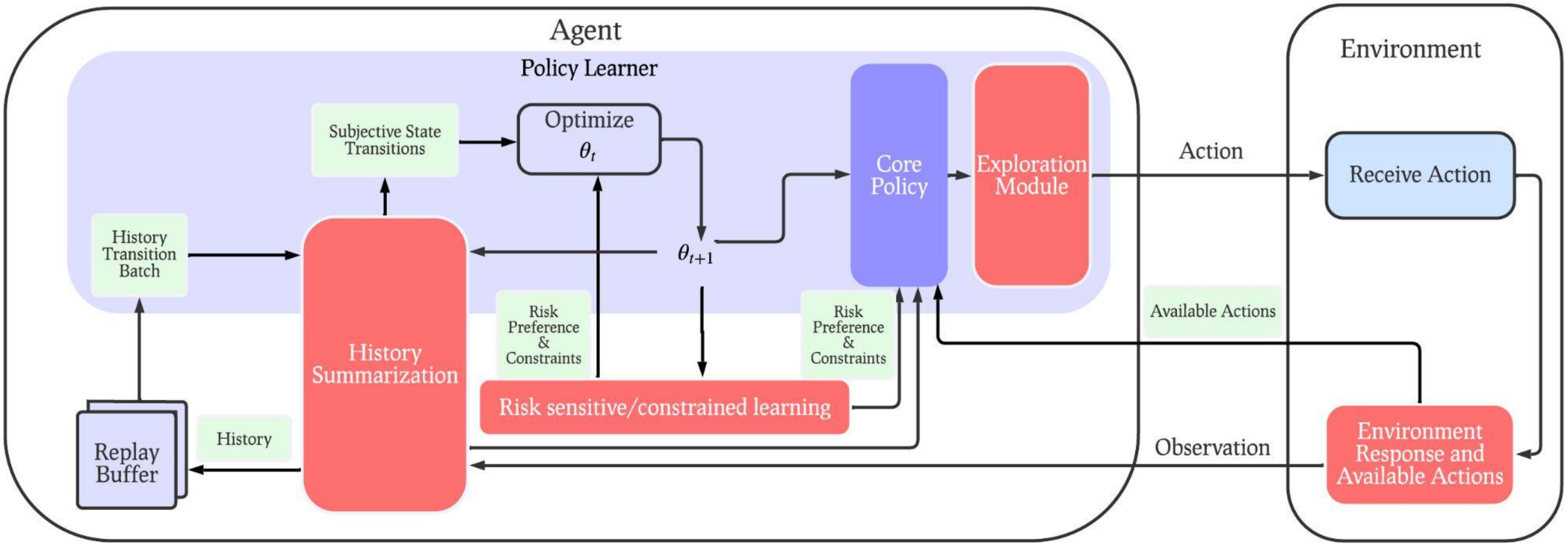
Challenges		Recommender Systems	Auction bidding	Creative Selection	Robotics	
Sparse Reward		✓	✓	✓	✓	
Dangerous and Risky State and Actions			✓		✓	
Partial Observability		✓	✓		✓	
Changing Action Space		✓		✓	✓	
Offline Learning		✓	✓	✓	✓	

Coming Back to Our Real-life Examples

Challenges		Recommender Systems	Auction bidding	Creative Selection	Robotics	Supply Chain
Sparse Reward		✓	✓	✓	✓	
Dangerous and Risky State and Actions			✓		✓	✓
Partial Observability		✓	✓		✓	✓
Changing Action Space		✓		✓	✓	✓
Offline Learning		✓	✓	✓	✓	✓

Coming Back to Our Real-life Examples

Challenges	RL Features	Recommender Systems	Auction bidding	Creative Selection	Robotics	Supply Chain
Sparse Reward	Online Exploration	✓	✓	✓	✓	
Dangerous and Risky State and Actions	Safety		✓		✓	✓
Partial Observability	History Summarization	✓	✓		✓	✓
Changing Action Space	Dynamic Action Space Support	✓		✓	✓	✓
Offline Learning	Offline RL	✓	✓	✓	✓	✓



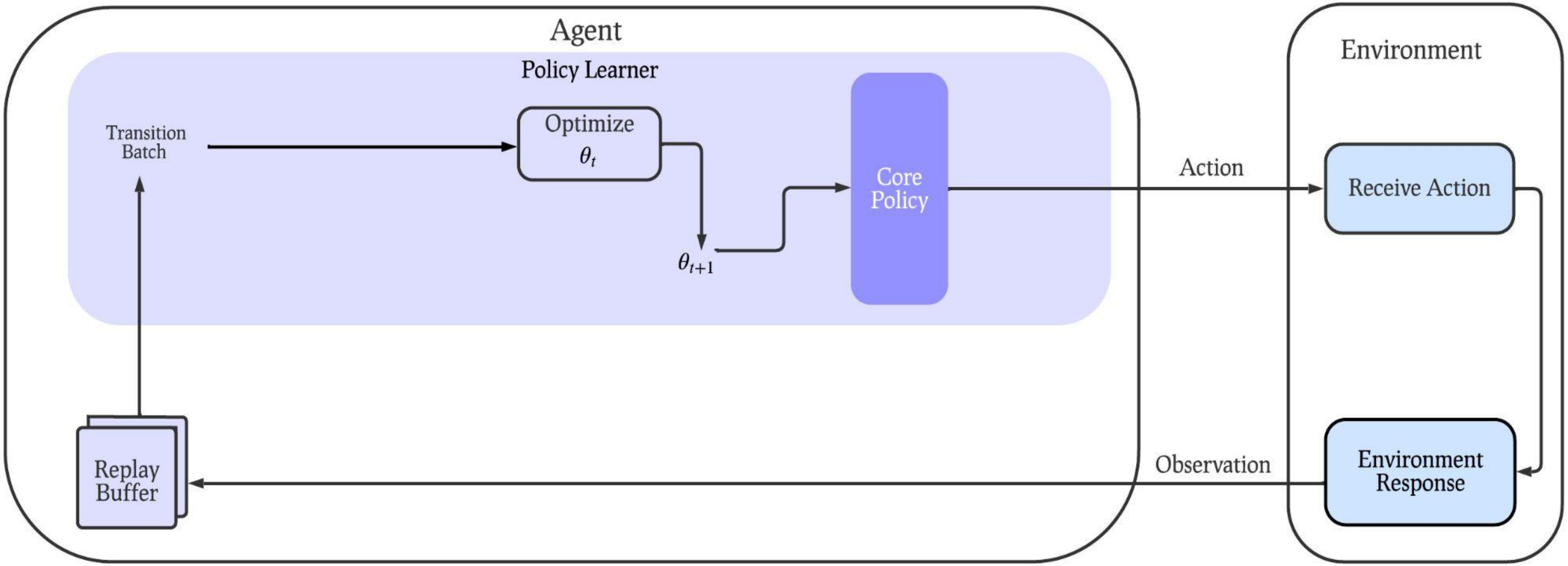
Agent
Policy Learner
Subjective State Transitions
Optimize
θ_t
Core Policy
Exploration Module
History Transition Batch
θ_{t+1}
History Summarization
Risk Preference & Constraints
Risk Preference & Constraints
Risk sensitive/constrained learning
Replay Buffer
History
Environment
Action
Receive Action
Available Actions
Observation
Environment Response and Available Actions

Compare Pearl to Existing Libraries

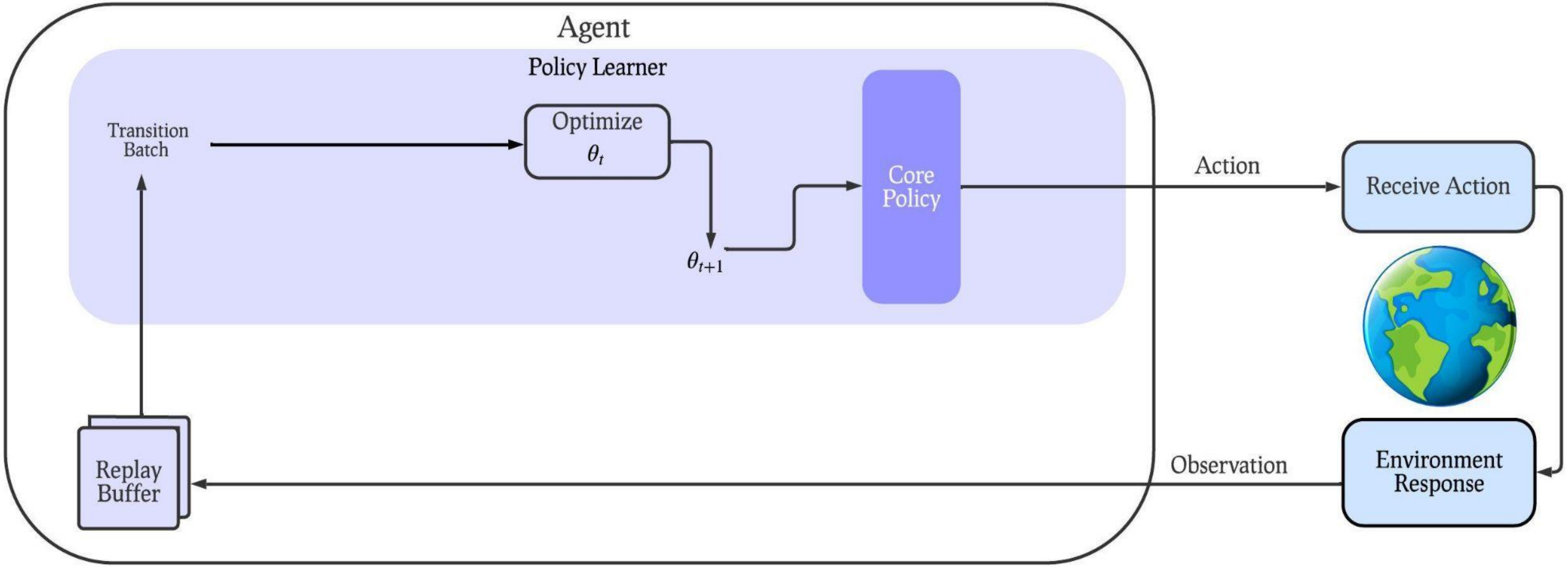
Pearl Features	Pearl	ReAgent (Superseded by Pearl)	RLLib	SB3	Tianshou	Dopamine
Agent Modularity	✅	❌	❌	❌	❌	❌
Dynamic Action Space	✅	✅	❌	❌	❌	❌
Offline RL	✅	✅	✅	✅	✅	❌
Intelligent Exploration	✅	❌	❌	❌	⚪ (limited support)	❌
Contextual Bandit	✅	✅	⚪ (only linear support)	❌	❌	❌
Safe Decision Making	✅	❌	❌	❌	❌	❌
History Summarization	✅	❌	✅	❌	⚪ (requires modifying environment state)	❌
Data Augmented Replay Buffer	✅	❌	✅	✅	✅	❌

06 | Pearl - Interface and Design

A Clear Interface between Agent and Environment



A Clear Interface between Agent and Environment



Step 1: Instantiate an agent

```
# assume an environment
observation_dim = env.observation_space.shape[0]
action_space = env.action_space


agent = PearlAgent(
    policy_learner=DeepQLearning(
        state_dim=observation_dim,
        action_space=action_space,
        hidden_dims=[64, 64],
    ),
    replay_buffer=FIFIOffPolicyReplayBuffer(
        size=100000
    ),
)
```

Step 2: Reset the agent state and action space

```
# optional in a product use case
observation, action_space = env.reset()

# sets the agent starting state and action_space
agent.reset(observation, acton_space)
```

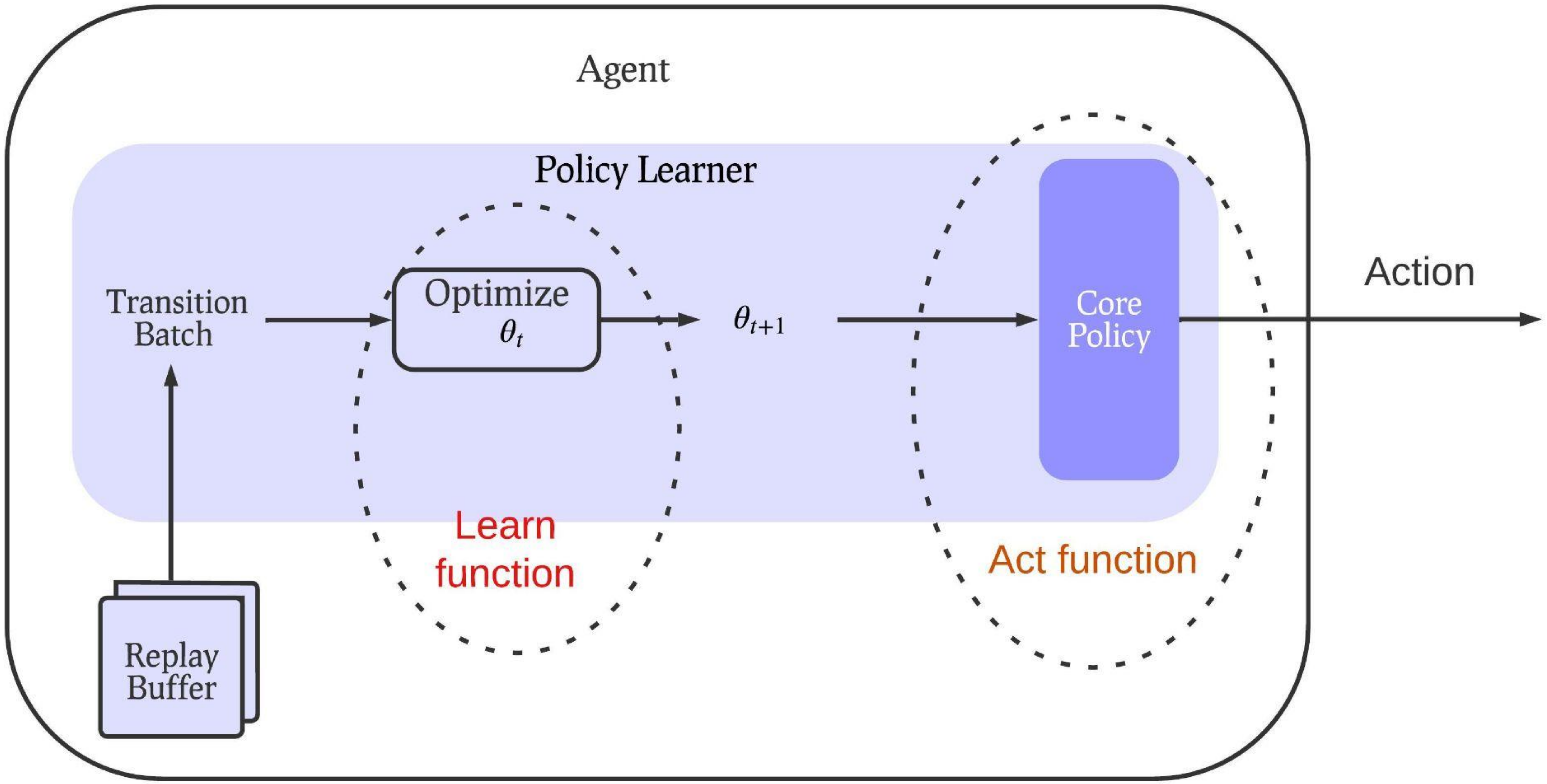
Step 3: Agent environment interaction

```
# agent takes an action given agent's state
action = agent.act()

# execute action and get feedback, optional in product use case
action_result = env.step(action)
```

Step 4: Agent stores the environment feedback in a replay buffer

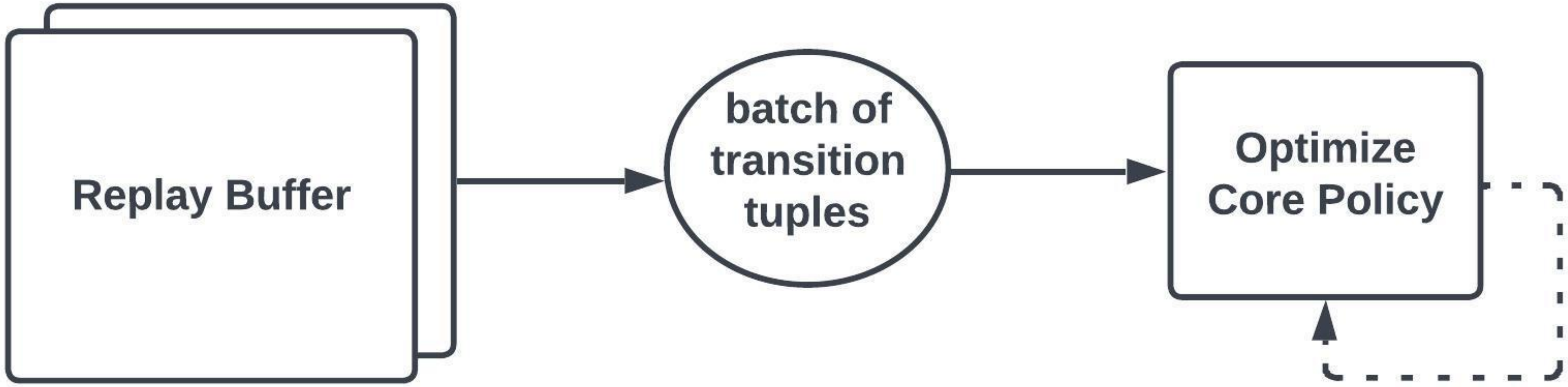
```
# pass feedback to agent
agent.observe(action_result)

def observe(self, action_result):

  # update the agent state to next observation and next action space
  self.state = action_result.observation
  self.action_space = action_result.action_space

  # create a transition tuple and store in replay buffer
  self.replay_buffer.push(
      observation=action_result.observation,
      reward=action_result.reward,
      ..,
  )
```

Policy Learning Module



Policy learner's learn function:

```
agent.learn()

def learn(self):

  # calls policy learner's learn function with the replay buffer
  self.policy_learner.learn(self.replay_buffer)
```



Policy learner’s act function:

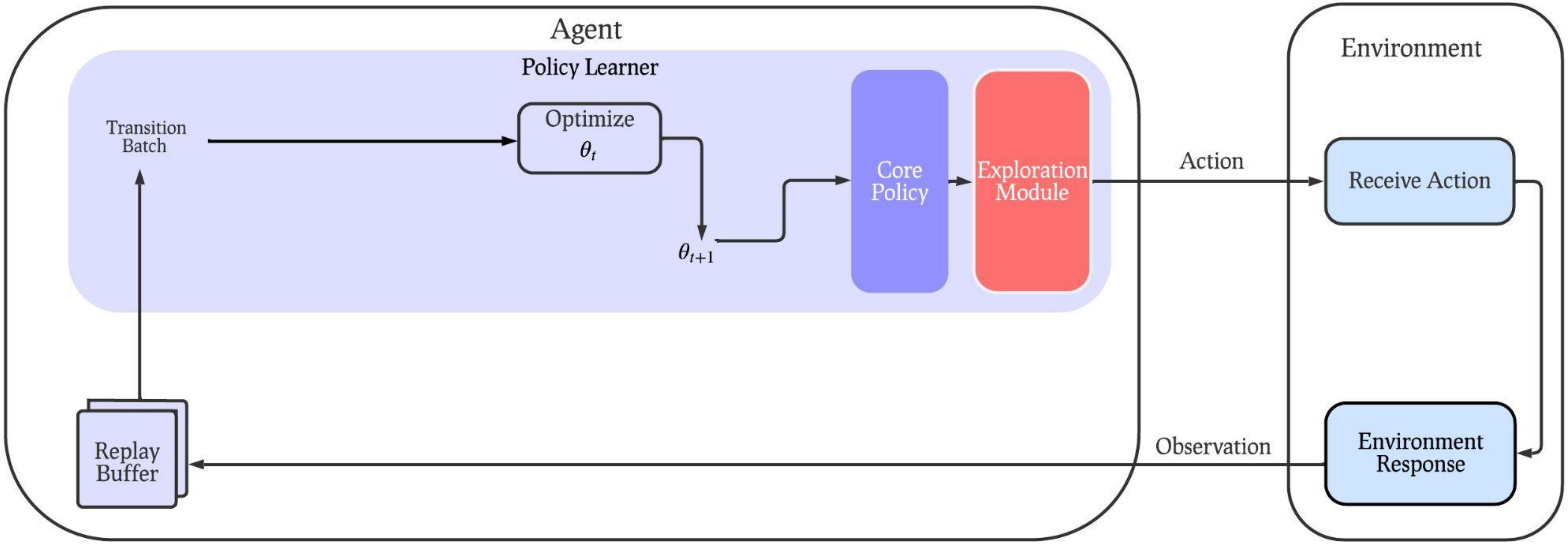
```
action = agent.act()

def act(self) -> Action:
   # calls policy learner’s act function with the
   # current agent state and action space
   action = self.policy_learner.act(
       state=self.state,
       action_space=self.action_space,
   )
```

We implement a suite of policy learning algorithms:

- **Actor-critic methods:**
  - Soft actor critic (SAC), Proximal policy optimization (PPO), REINFORCE
  - Deep deterministic policy gradients (DDPG), and twin delayed version (TD3)
- **Value-based methods:** Deep Q learning and variants
- **Distributional RL methods:** Quantile regression based deep q learning
- **Offline RL methods:** Conservative Q learning, Implicit Q learning
- **Bandit learning:** Neural and linear bandit algorithms

Exploration Module

- Exploration module is attached to the policy learner module for structured exploration.



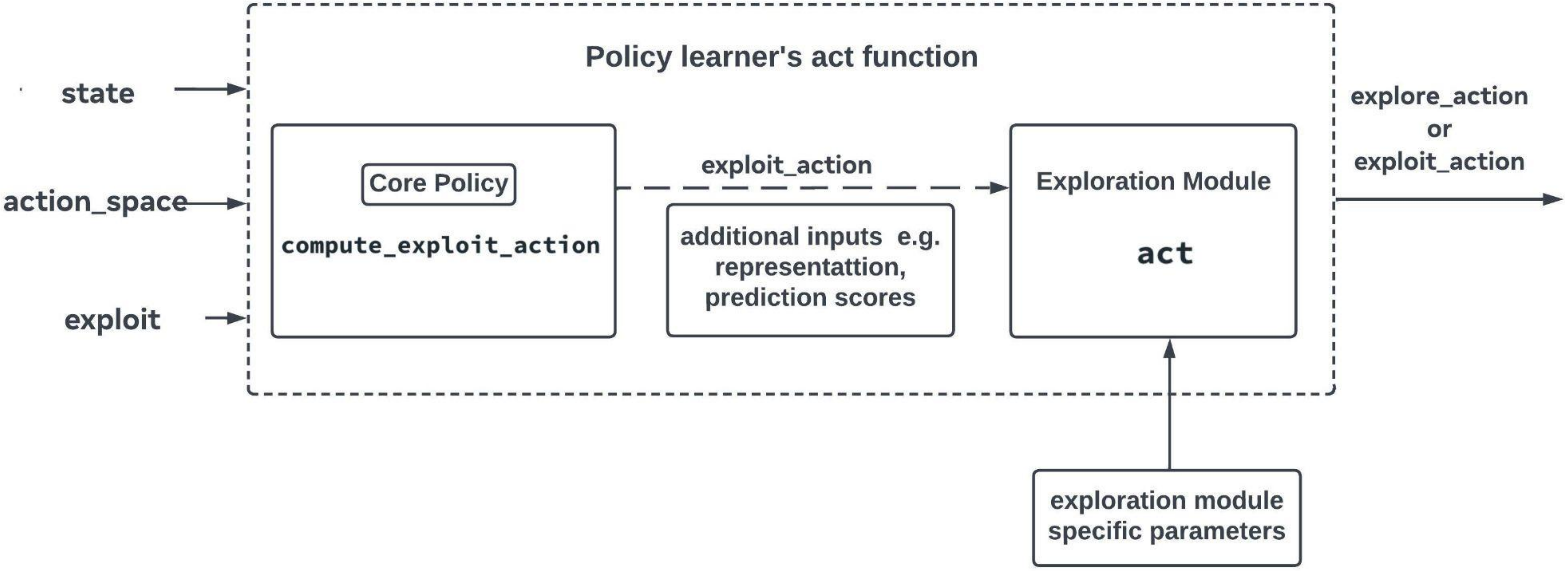
Instantiate a Pearl agent with exploration module

```
agent = PearlAgent(
    policy_learner=DeepQLearning(
        state_dim=observation_dim,
        action_space=action_space,
        hidden_dims=[64, 64],
        exploration_module=EGreedyExploration(epsilon=0.05)
    ),
    replay_buffer=FIFIOffPolicyReplayBuffer(
        size=100000
    ),
)
```

Act function of the agent:

```
# toggle on/off exploration (e.g. learning vs deployment)
do_exploit = False
action = agent.act(exploit=do_exploit)


def act(self, exploit: bool = False) -> Action:
  # calls policy learner's act function with the
  # current agent state and action space
  action = self.policy_learner.act(
      state=self.state,
      action_space=self.action_space,
      exploit=exploit,
  )
```

Act function of the policy learner uses the exploration module



We implement different algorithms:

❖ Exploration for (neural) bandit learning:

- Upper confidence bound (UCB) based exploration,
- Thompson sampling,
- Square CB

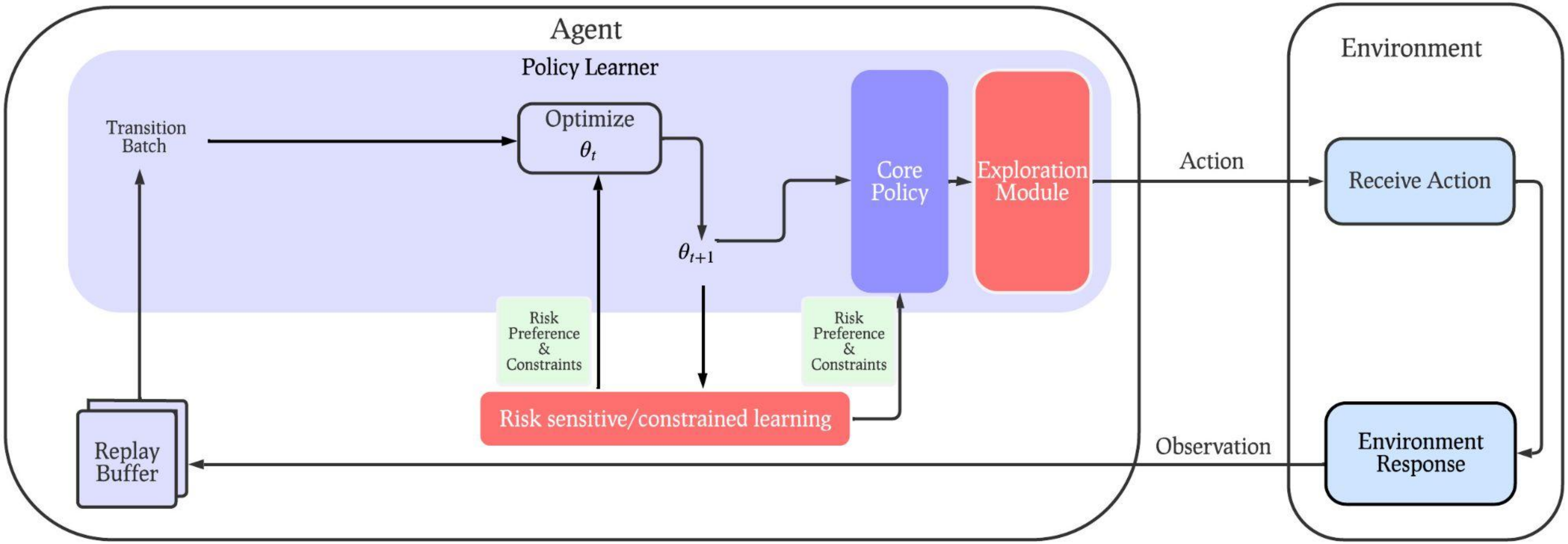
❖ Exploration for learning in sequential-decision making:

- Ensemble based deep exploration
- Epsilon greedy, for discrete action space
- Gaussian random exploration for continuous actions
- Propensity based exploration for stochastic actors

Safety Module

A set of two different submodules that can:

- enable risk-sensitive learning,
- enable constrained policy optimization

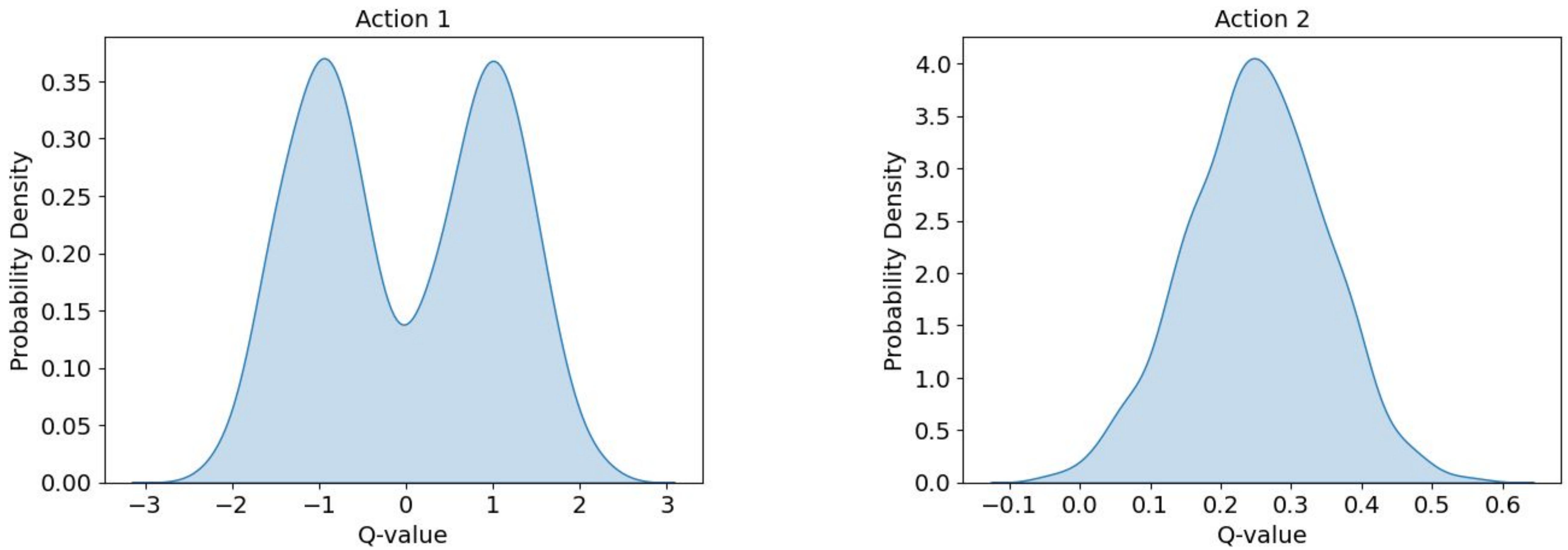


Risk sensitive learning with distributional policy learners

```
agent = PearlAgent(
    policy_learner=QuantileRegressionDeepQLearning(
        state_dim=observation_dim,
        action_space=action_space,
        hidden_dims=[64, 64],
        exploration_module=EGreedyExploration(epsilon=0.05),
    ),
    safety_module=QuantileNetworkMeanVarianceSafetyModule(
        variance_weighting_coefficient: float = 0.2,
    ),
    replay_buffer=FIFIOffPolicyReplayBuffer(
        size=100000
    ),
)
```

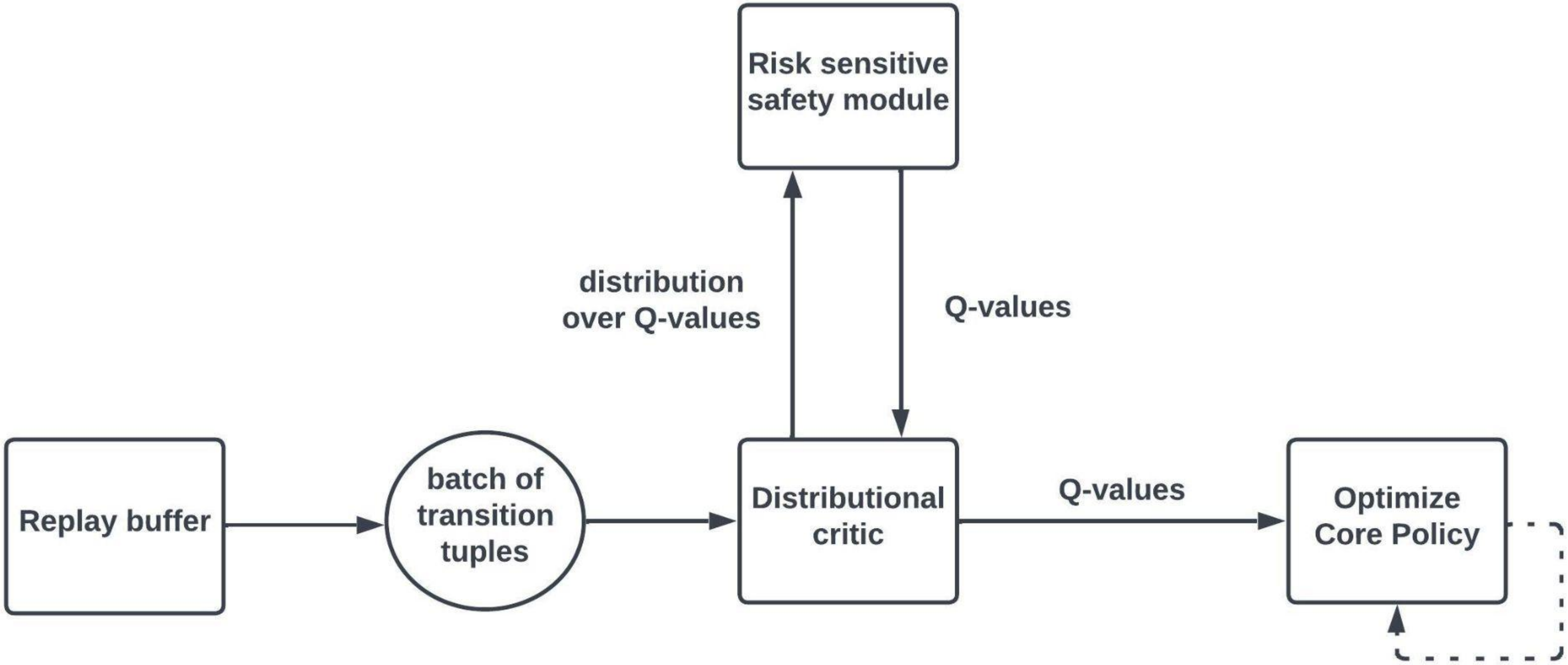
Distributional policy learning models a distribution over Q values:

- Capture uncertainty in Q-value functions due to stochasticity in the MDP (randomness of reward and transition probabilities).

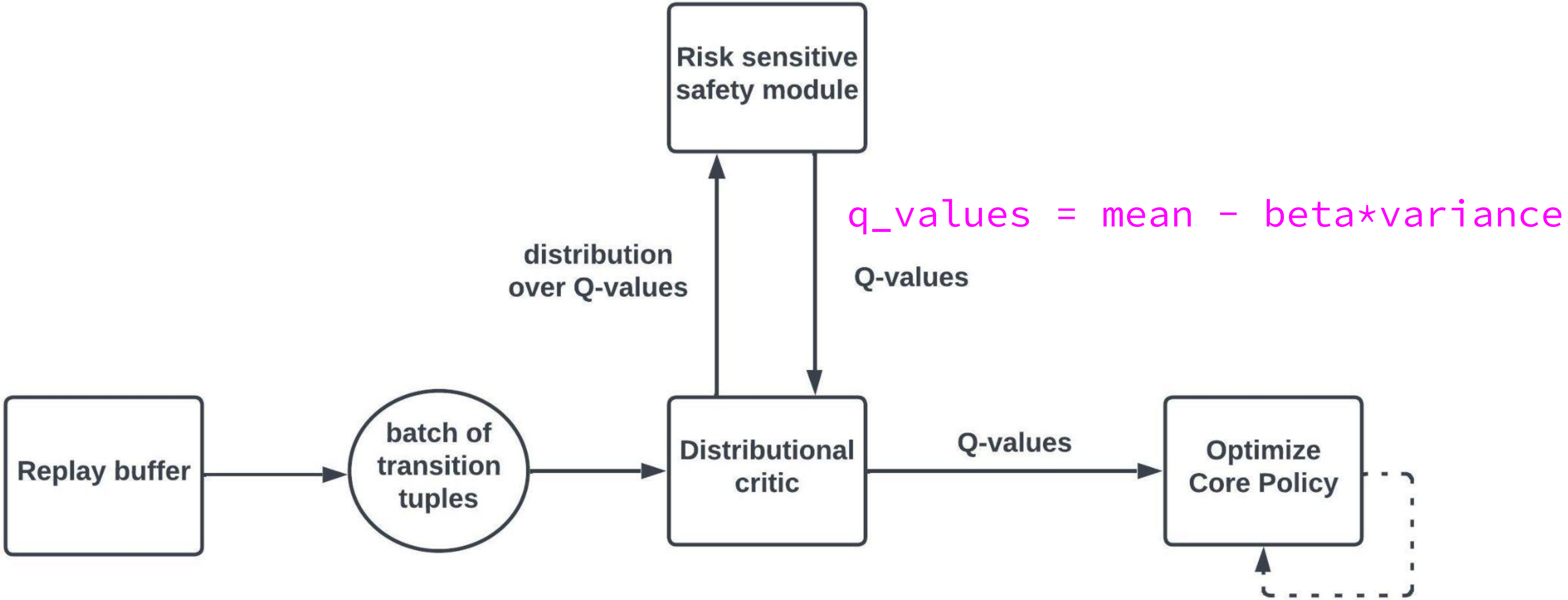


- Implicit Quantile Networks (Dabney et. al. 2018), Quantile Regression Deep Q learning (Dabney et. al . 2017) compute a quantile approximation to the return distribution.

Policy learning with risk sensitive safety module



Policy learning with risk sensitive safety module



Constrained policy optimization safety module

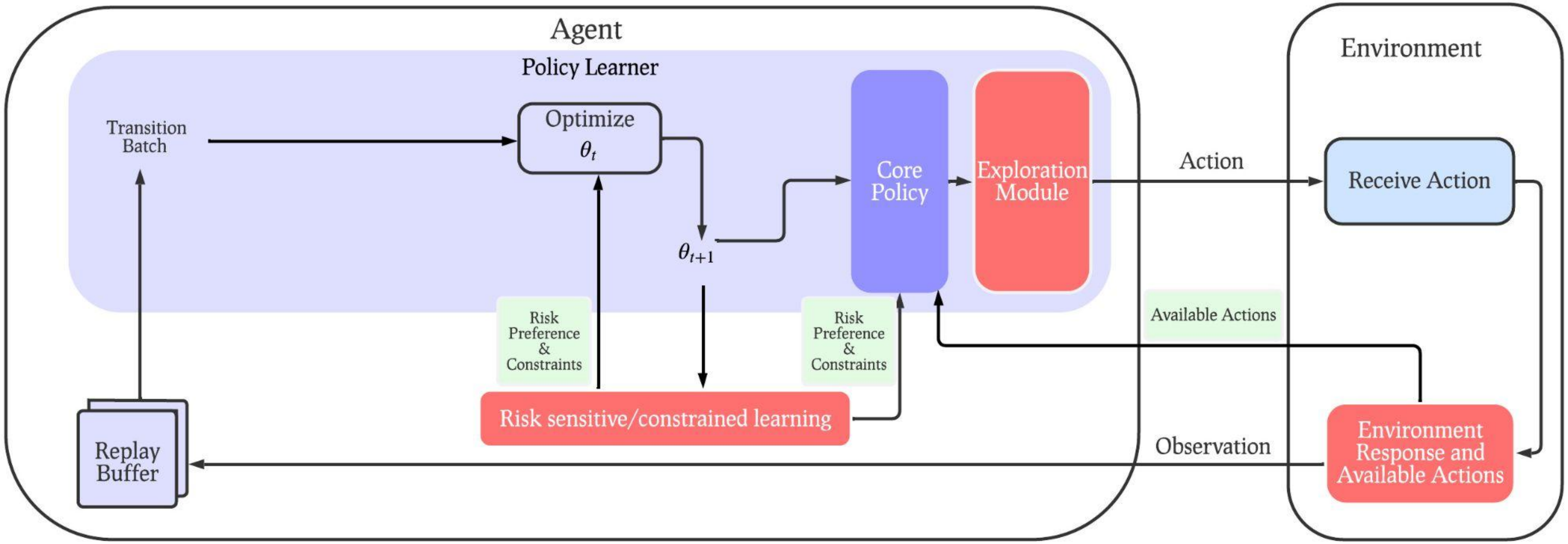
- Enables learning in constrained sequential decision making problems.
- Every state action has a cost in addition to the reward, i.e. `r(s,a) and c(s,a).`
- Maximize cumulative rewards, subject to cumulative costs being bounded

$$\mathbb{E}_\pi\left[\sum_t \gamma^t c(s_t, a_t)\right] \leq \alpha,$$

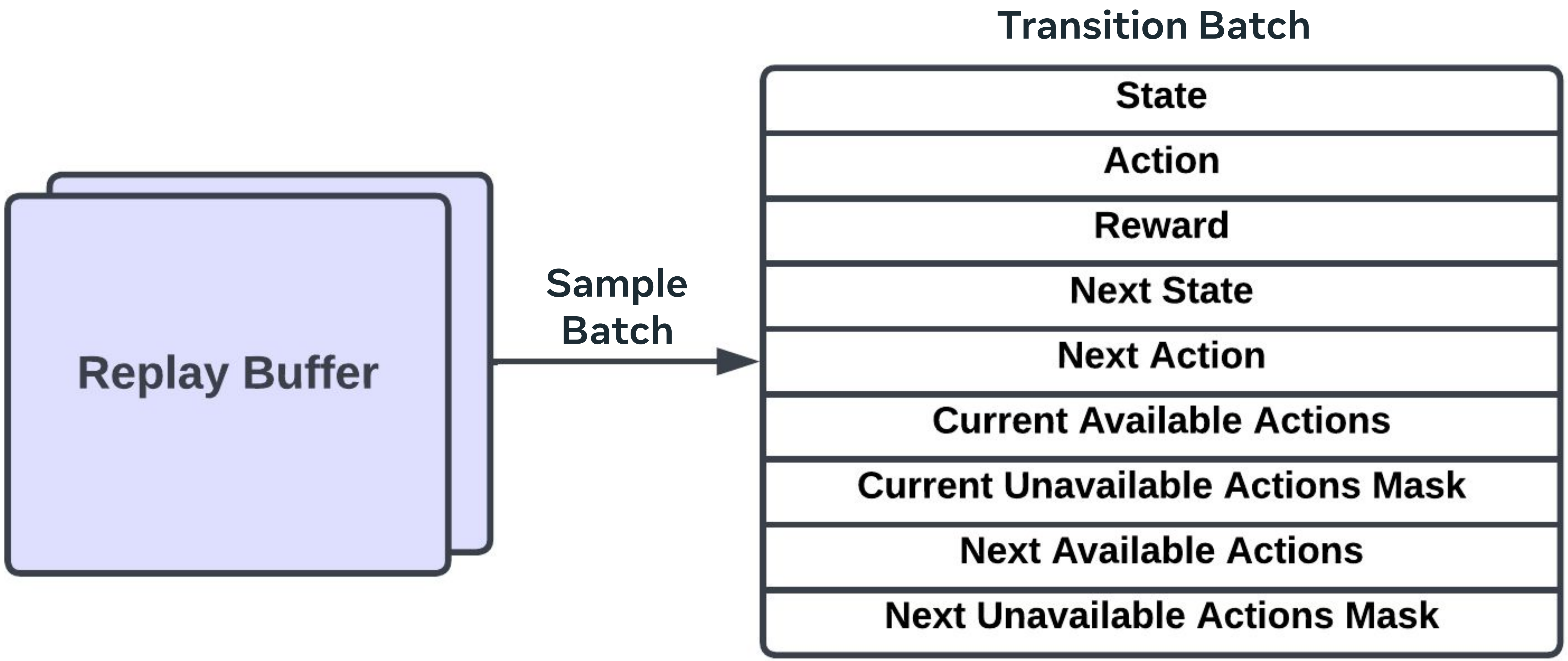
- We implement Reward Constrained Policy Optimization (Tessler et. al, 2019) which can be used with different policy learners.

Dynamic Action Spaces

- Many real world problems (like recommender systems) require working with dynamic action spaces.
- We enable Pearl to handle discrete dynamic action spaces.



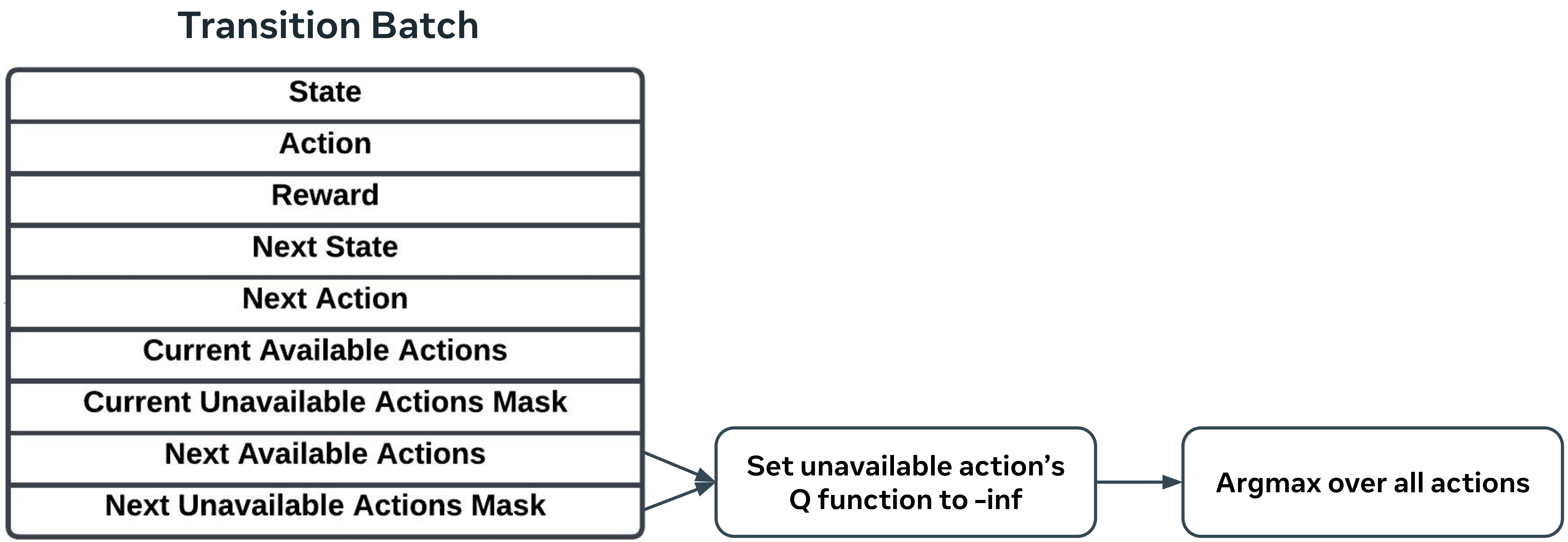
Dynamic Action Spaces: Replay Buffer Design



Dynamic Action Spaces: Replay Buffer Design

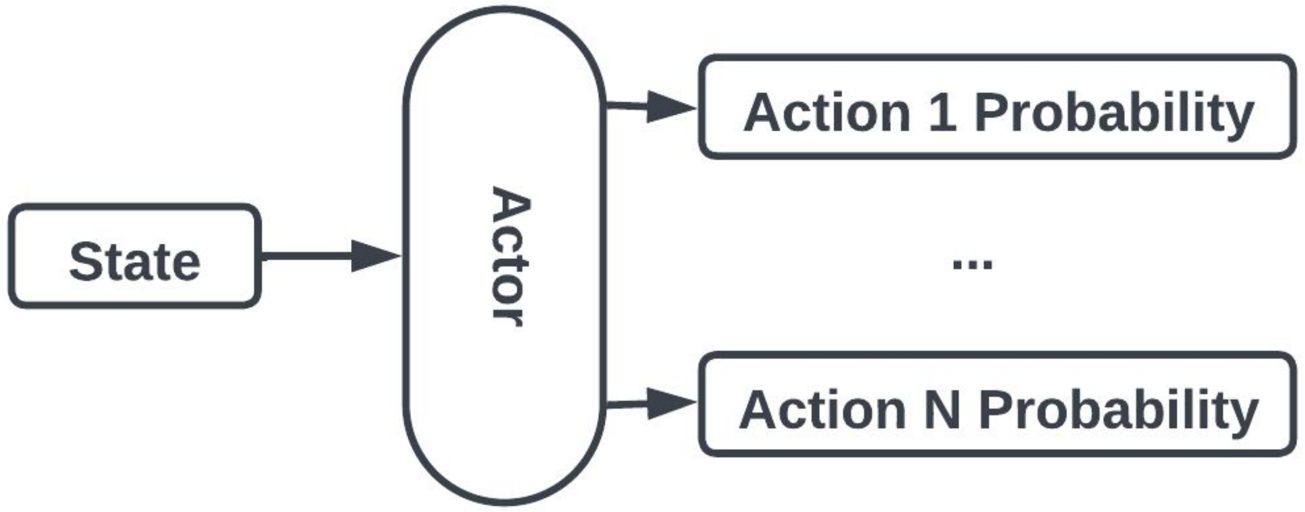
```
Example: Maximum number of 5 actions,
        2 available actions, [0, 6]

        available_actions_tensor_with_padding = [
            [0],
            [6],
            [0],
            [0],
            [0],
        ]
        unavailable_actions_mask = [0, 0, 1, 1, 1]
```

Dynamic Action Spaces: Value-Based Model Design



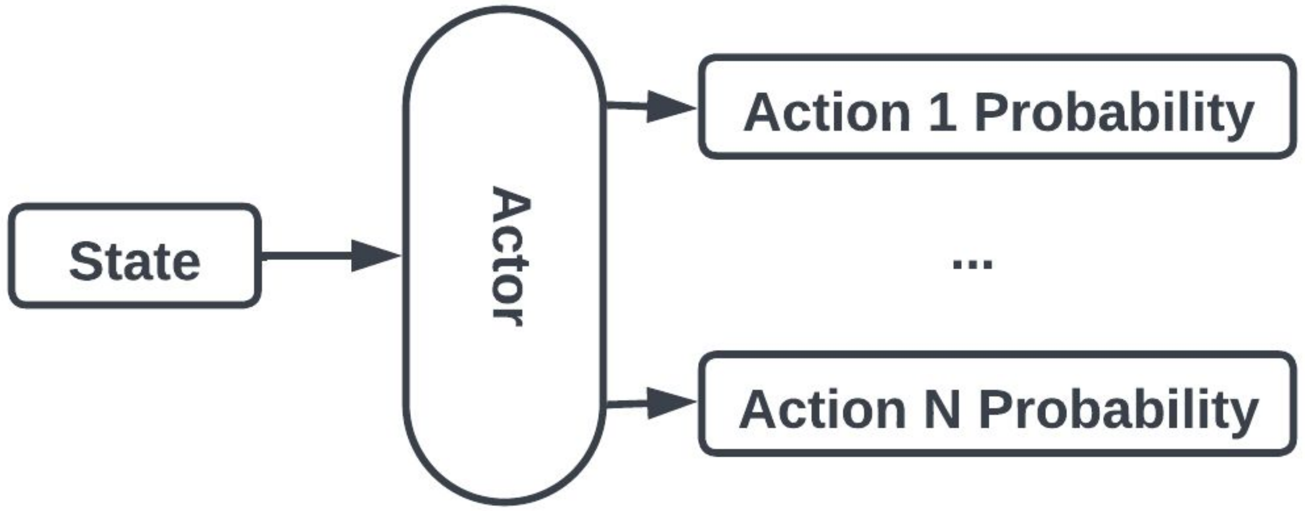
Dynamic Action Spaces: Dynamic Action Actor Neural Network

- Traditional Actor Neural Network

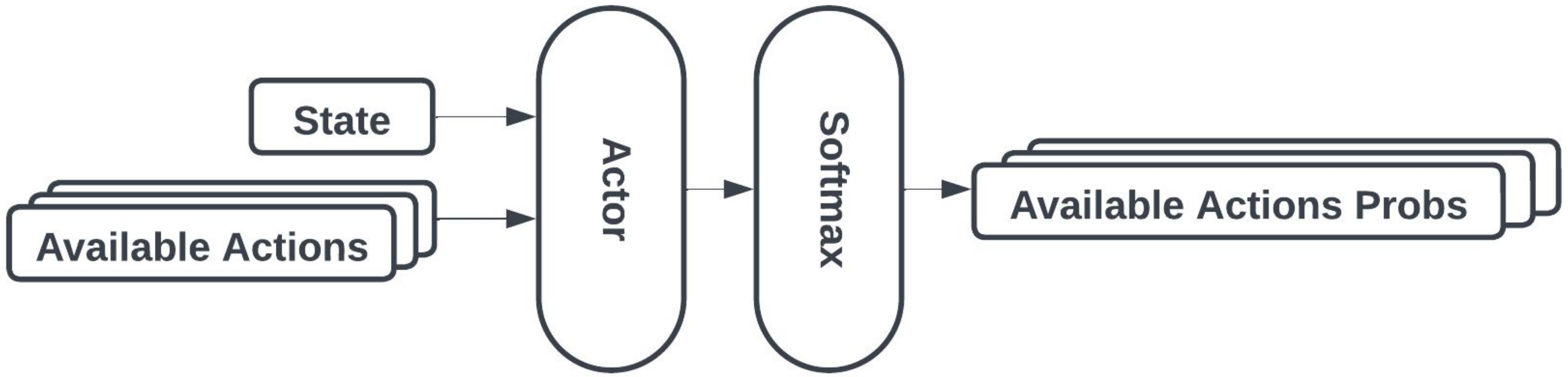


Dynamic Action Spaces: Dynamic Action Actor Neural Network

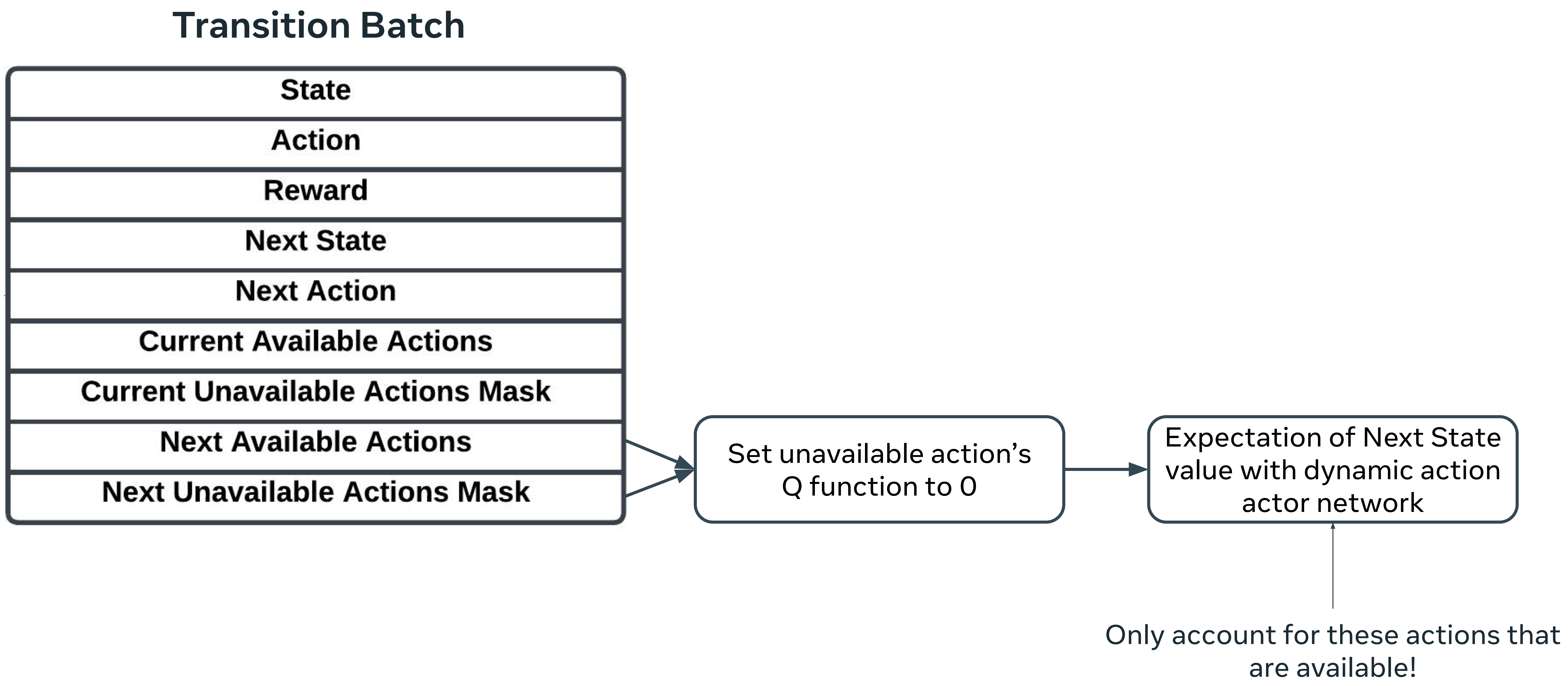
- Traditional Actor Neural Network



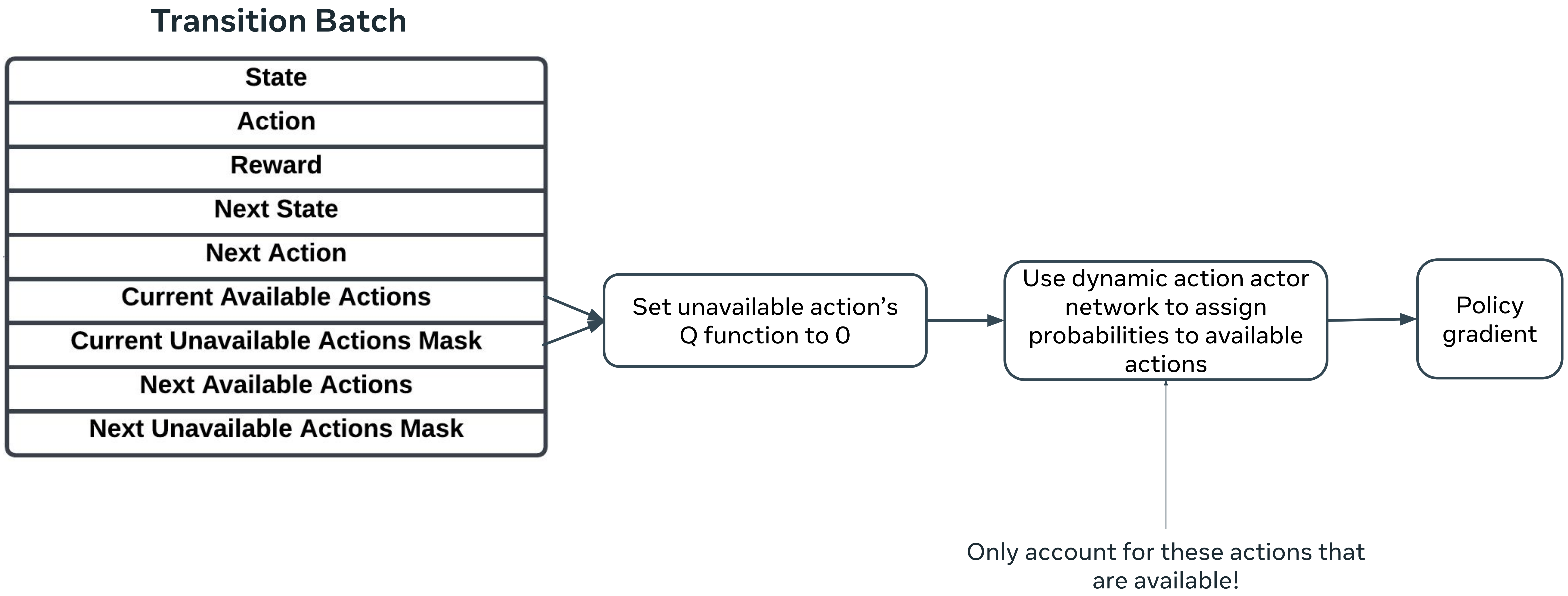
- Dynamic Action Actor Neural Network



Dynamic Action Spaces: Actor-Critic Design (learn_critic)

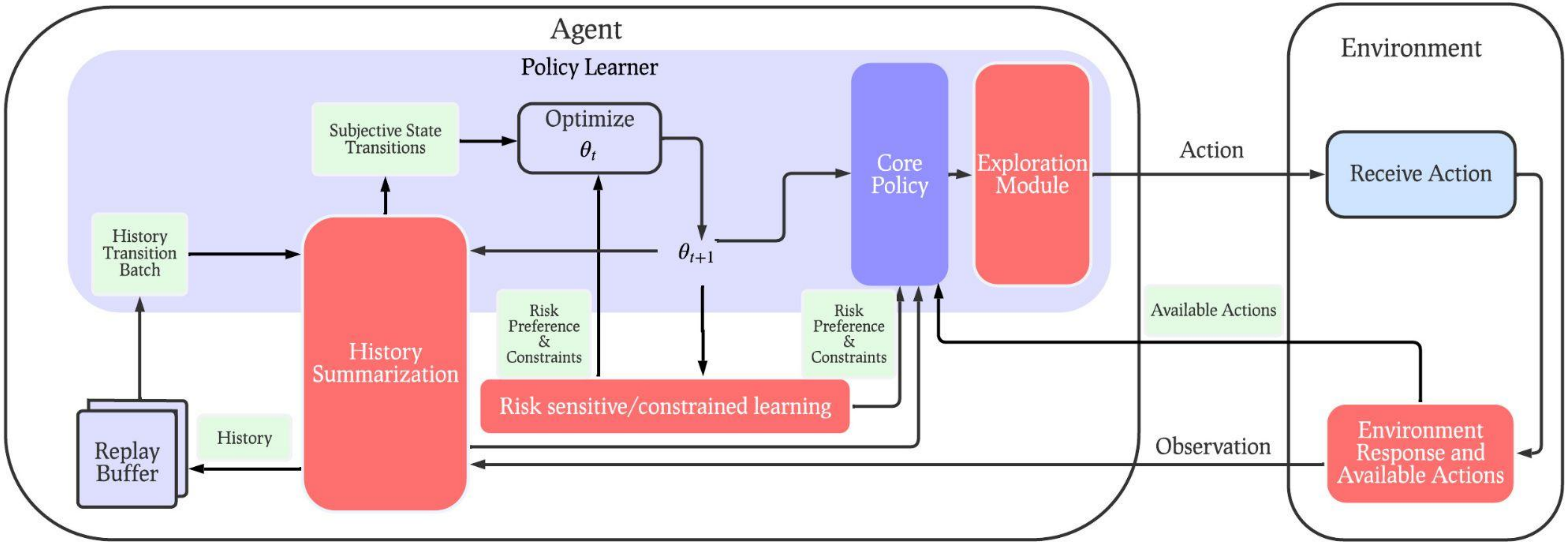


Dynamic Action Spaces: Actor-Critic Design (learn_critic)



History Summarization Module

- A history summarization module enables learning in partially observable environments.



History Summarization with LSTM

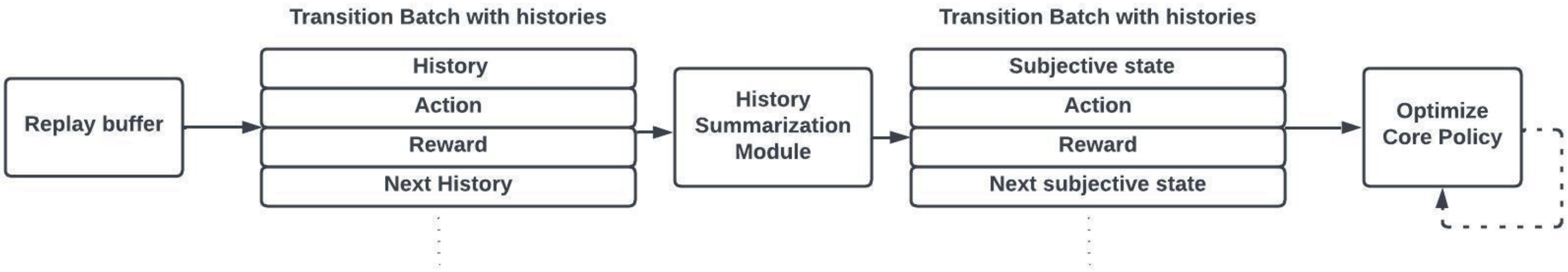
```
agent = PearlAgent(
    policy_learner=DeepQLearning(
          state_dim=observation_dim,
          action_space=env.action_space,
          hidden_dims=[64, 64],
          exploration_module=EGreedyExploration(epsilon=0.05),
    ),
    history_summarization_module=LSTMHistorySummarizationModule(
          observation_dim=observation_dim,
          action_dim=action_dim,
          hidden_dim=128,
          history_length=8,
    ),
    ...
)
```

Histories instead of observations are stored in the replay buffer

```
# agent's observe function
def observe(self, action_result):

    # get current history
    current_history = self.history_summarization_module.get_history()

    # update history using the latest observation and action
    self.history_summarization_module.update_history(
        action_result.observation,
        action_result.action,
    )
    new_history = self.history_summarization_module.get_history()

    # store histories instead of observations in the replay buffer
    self.replay_buffer.push(state=current_history, .. )
```

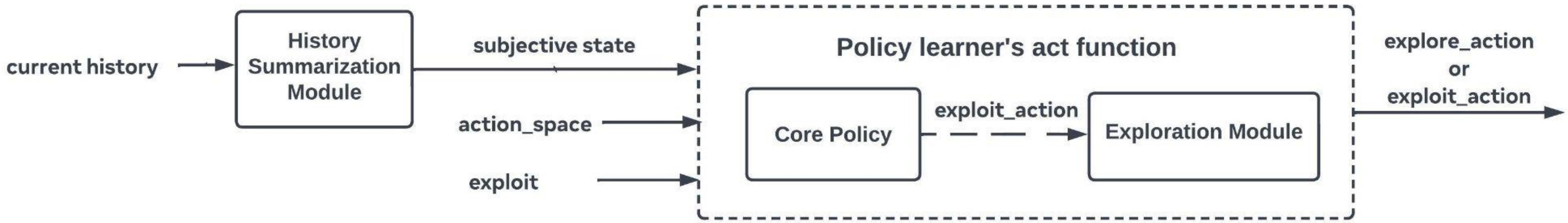
Histories are summarized into a subjective state during policy learning

```
# batch of histories summarized to batch of subjective state
# by doing a forward pass through history summarization module

batch.state = history_summarization_module.summarize_history(
      history=batch.state,
)
batch.next_state = ..
```



Histories are summarized during agent-environment interaction

```
# get current history
# compute agent's subjective state by summarizing current history
# subjective state passed to policy learner's act function
```



```
agent = PearlAgent(
    policy_learner=QuantileRegressionDeepQLearning(
            state_dim=observation_dim,
            action_space=env.action_space,
            hidden_dims=[64, 64],
            exploration_module=EGreedyExploration(epsilon=0.05),
    ),
    history_summarization_module=LSTMHistorySummarizationModule(
                observation_dim=observation_dim,
                action_dim=action_dim,
                hidden_dim=128,
                history_length=8,
    ),
    safety_module=QuantileNetworkMeanVarianceSafetyModule(
            variance_weighting_coefficient=0.1
    ),
    replay_buffer=FIFIOffPolicyReplayBuffer(
            size=100000
    ),
)
```

08 | Summary

Summary

- Pearl is a Reinforcement Learning AI Agent Library that adapts to many real-world sequential decision making challenges
- Pearl's modular design offers researchers and practitioners an easy means to combine multiple reinforcement learning features into a single agent
- Pearl's native pytorch support and clean interface allows easy product deployment

Pearl Github Repo:

github.com/facebookresearch/pearl

Meta

Meta