arXiv:submit/5279090 [cs.AI] 6 Dec 2023

Pearl: A **P**roduction-R**ea**dy **R**einforcement **L**earning Agent

Zheqing Zhu*, Rodrigo de Salvo Braz, Jalaj Bhandari, Daniel Jiang, Yi Wan, 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

December 6, 2023

Abstract

Reinforcement Learning (RL) offers a versatile framework for achieving long-term goals. Its generality allows us to formalize a wide range of problems that real-world intelligent systems encounter, such as dealing with delayed rewards, handling partial observability, addressing the exploration and exploitation dilemma, utilizing offline data to improve online performance, and ensuring safety constraints are met. Despite considerable progress made by the RL research community in addressing these issues, existing open-source RL libraries tend to focus on a narrow portion of the RL solution pipeline, leaving other aspects largely unattended. This paper introduces **Pearl**, a **P**roduction-r**ea**dy **RL** agent software package explicitly designed to embrace these challenges in a *modular* fashion. In addition to presenting preliminary benchmark results, this paper highlights Pearl's industry adoptions to demonstrate its readiness for production usage. Pearl is open sourced on Github at github.com/facebookresearch/pearl and its official website is located at pearlagent.github.io.

Keywords: Reinforcement learning, open-source software, python, pytorch

1 Introduction

The field of reinforcement learning (RL) has achieved significant successes in recent years. These accomplishments encompass a range of achievements, from surpassing human-level performance in Atari Games (Mnih et al., 2015) and Go (Silver et al., 2017), to controlling robots to in complex manipulation tasks (Mnih et al., 2015; Peng et al., 2018; Levine et al., 2016). Moreover, the practical applications of these advancements extend into real-world systems, including recommender systems (Xu et al., 2023) and large language models (Ouyang et al., 2022). In addition to these successful RL systems, significant progress has been made in designing open-resource libraries that enable developing RL systems easily. These libraries include RLLib (Liang et al., 2018), Stable-Baselines 3 (Raffin et al., 2021), and Tianshou (Weng et al., 2022), to name a few.

In addition to tackling the core issues of delayed rewards and downstream consequences, successful RL agents must address several significant challenges. One of them is the delicate balance between exploration and exploitation. An RL agent must actively engage in exploration to gather information about actions and their outcomes. This challenge is compounded by the fact that the environment may not always offer complete transparency regarding its internal state, requiring the agent to infer the current state from its interaction history. In order to avoid catastrophic situations or accommodate other preferences, an RL agent may also need to incorporate additional constraints, such as safety considerations or risk requirements, throughout the course of learning.

While the importance of these challenges is widely acknowledged by the RL community, existing open source RL libraries often do not address them adequately. For example, important features like exploration, safe/constrained policy learning, credit assignment for long-horizon delayed-reward

[*]Corresponding author. Please email: billzhu@meta.com

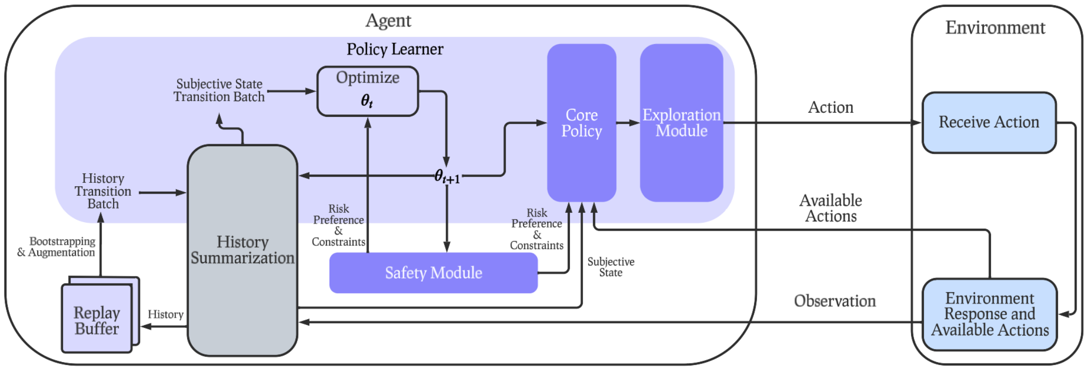


Figure 1: Pearl Agent Interface

settings, and partial observability are frequently absent. In addition, many libraries do not include offline RL methods, even if these methods are commonly adopted in real-world applications Moreover, the open source community has typically viewed RL and bandit problems as two distinct settings with separate codebases. We offer a detailed discussion about existing libraries in Section 3.

In this paper, we introduce `Pearl`, a **P**roduction-R**ea**dy **R**einforcement **L**earning Agent, an open-source software package, which aims to enable users to build a versatile RL agent for their real-world applications. The focal point of the package is a `PearlAgent`, which, in addition to a main (offline or online) policy learning algorithm, encapsulates one or more of the following capabilities: intelligent exploration, risk-sensitivity, safety constraints, and history summarization for the partially-observed/non-Markovian setting. Our package includes several recent algorithmic advancements that address these challenges in the RL research community. Augmenting an RL agent with these capabilities is essential for both research and improving adoption of RL for real-world applications. To achieve these capabilities, we adopted a fully modular design philosophy, empowering researchers and practitioners to tailor and combine the features their agents employ as they see fit. For example, `PearlAgent` offers a unified implementation of both RL and bandit methods.

`Pearl` is built on native PyTorch to support GPU and distributed training. It also provides a suite of utilities for testing and evaluation. `Pearl` is currently adopted by multiple industry products, including recommender systems, ads auction pacing, and contextual-bandit based creative selection. These applications require support from `Pearl` across online exploration, offline learning, safety, data augmentation, history summarization, and dynamic action spaces.

This paper serves as an introduction of our motivation, features and design choices for `Pearl`, and simple illustrations of user interface to the community. More details are given in Section 2. Section 3 compares `Pearl` to other open-source RL libraries. An initial set of benchmarking results is presented in Section 4. Section 5 details current industry adoptions of `Pearl`.

2 Pearl Agent

This section gives an overview of the design of `PearlAgent`. `PearlAgent` has five main modules, namely, `policy_learner`, `exploration_module`, `history_summarization_module`, `safety_module` and `replay_buffer`. To facilitate a better understanding of the these modules, we will use the following notations in throughout the rest of the paper:

1. Observation: O_t denotes the observation the agent receives at time t. This can be a Markovian state, a non-Markovian partial observation, or a context in the contextual bandit setting.

2. Action: $A_t \in \mathcal{A}_t$ denotes an action the agent chooses at time t, while $\mathcal{A}_t$ denotes the available action space at time t. We subscript action space by time to enable dynamic action spaces, an important feature of real-world applications (e.g., in recommender systems, the set of available actions changes with time).

3. Reward: $R_t \in \mathbb{R}$ indicates a scalar reward the agent receives at time step t. In this work, we assume that when an agent takes an action at time t, it receives a reward at time $t + 1$.

4. Markovian state and history: In a Markovian environment, the observation O_t is equivalent to the Markovian state $S_t \in \mathcal{S}$. When the environment is partially observable, we define history $H_t = (O_0, A_0, R_1, O_1, A_1, \ldots, O_t, A_t)$ to denote the history of interactions.

5. Interaction tuple: $\mathcal{E}_t = (S_t, A_t, R_{t+1}, S_{t+1}, \mathcal{A}_{t+1})$ indicates a tuple of current state, action, reward, next state and action space at the next time step. In the case of a contextual bandit problem, S_{t+1} and $\mathcal{A}_{t+1}$ can be thought of as set to None.

2.1 Agent Design

Consider the following typical usage scenario: A user of `Pearl` has access to offline data, either in the form of environment interaction records or (partial) trajectories, along with the ability to interact with the environment to gather additional online data. In designing the `PearlAgent`, we prioritize several key elements that are essential for efficient learning in practical sequential decision-making problems. Together, they serve as essential building blocks of a comprehensive RL agent:

1. **Offline learning/pretraining**: Depending on the problem setting (contextual bandit or Markovian transitions), an RL agent should be able to leverage an offline learning algorithm to learn and evaluate a policy.

2. **Online learning**: With a pretrained/prior policy, the agent should be able to a) explore to intelligently collect the most informative interaction tuples, and b) learn from the collected experiences to reason about the optimal policy. The agent should have access to specialized policy optimization algorithms appropriate for different problem settings.

3. **Safe learning**: For both offline and online learning, an RL agent should have the ability to incorporate some form of safety or preference constraints. Users might want to impose such constraints both for data collection (in the online setting) as well as for policy learning.

4. **Representation learning and history summarization**: In addition to different modes of learning, the agent should be able to leverage different models for learning state representations, value and policy functions. Moreover, for partially observable environments, it is important for the agent to have the ability to summarize histories into state representations.

5. **Replay Buffers**: For efficient learning, an RL agent should have the ability to reuse data efficiently and subset the environment interaction data which it prioritizes to learn from. A common way to do this is through the use of a replay buffer, customized to support different problem settings. To enhance learning efficiency, it is important for the agent to have the flexibility to augment the replay buffer with auxiliary information (say, for credit assignment).

`Pearl` supports all of the above features in a unified way.[1] Besides a suite of policy learning algorithms, users can instantiate a `PearlAgent` to include an appropriate replay buffer, a history summarization module[2] to learn from non-Markovian transitions as well as a safe learning module to account for preferences/constraints during policy learning and to filter out undesirable actions during collection of new environment interactions. Modular code design enables seamless integration between the different functionalities in a `PearlAgent`. Figure 1 visualizes different components of a `PearlAgent` and how they interact with each other.

[1] For this iteration, we plan to only support model-free RL methods. Offline evaluations, and model based RL methods are planned for the next version of `Pearl`

[2] We are working to integrate more general state representation tools in `Pearl` and hope to include it in this version's code release.

2.1.1 Agent Interface

Figure 1 illustrates interactions amongst components of a `PearlAgent` in an online learning paradigm. Each learning epoch alternates between getting a new environment interaction and a training pass. Starting from an observation O_t, along with an estimate of the policy π_t, the `PearlAgent` queries for an interaction tuple $\mathcal{E}_t$ by taking action A_t. Note that in all the discussed `Pearl` components below, `Pearl` is not confined to a static action space; it is capable of adapting to dynamic action spaces that evolve over time.

To account for the trade-off between exploration and exploitation, `PearlAgent` decides to take action A_t by querying its `exploration_module` (which outputs an exploratory action A^{explore}), in conjunction with the `policy_learner` (which outputs an exploit action A^{exploit}). To compute the exploit action $A_t^{\text{exploit}} = \pi_t(S_t)$, `PearlAgent` enables interaction between the `policy_learner` and the `history_summarization_module`, which outputs the state representation.[3] `PearlAgent` design enables the `safety_module` to interact with both the `policy_learner` and `exploration_module` and account for safety constraints (for example, to filter out undesirable subset of actions)[4] when computing A^{explore} and A^{exploit} respectively. The interaction tuple $\mathcal{E}_t$ is stored in the `replay_buffer`.

During a training round at time t, a batch of interaction tuples are fetched from the `replay_buffer`; `PearlAgent` then queries the `history_summarization_module` to compute the corresponding state representations and generate a batch of history transitions $B_t = \{\mathcal{E}_k\}_{k=1}^{K}$. This batch of data tuples is used to update the `policy_learner`, accounting for safety and preference constraints specified by its `safety_module`. It is also used to update parameters of the `history_summarization_module`.

For an offline learning setup, readers can imagine the environment to be a dataset of interaction tuples and the exploration module to be inactive. Instead of querying for a new environment interaction tuple $\mathcal{E}_t$ by passing action A_t to the environment, an offline `PearlAgent` would simply query for one of the interaction tuple already present in the offline dataset.

2.1.2 Policy Learner

In `Pearl`, the `policy_learner` module implements different policy learning algorithms commonly used in RL. Any `policy_learner` module maintains the agent's current estimate of the optimal policy and updates it using a batch of interaction tuples. A `policy_learner` module interacts with an exploration module, since many forms of exploration use uncertainty estimates of the return[5] or action distribution (in the case of stochastic policies). We do this by implementing the `act` and `learn` method for policy learners in `Pearl`. For value based policy learners and actor-critic methods, the `learn` method is used to update the corresponding value function estimates. We list the different policy learners supported in `Pearl`.

- (Contextual) bandit algorithms: Common bandit learning methods involve reward modeling, using an `exploration_module` for efficient exploration.[6] `Pearl` supports Linear and Neural Bandit Learning along with different `exploration_modules`, as well as the SquareCB (Foster & Rakhlin, 2020) algorithm.

- Value-based methods: Deep Q-learning (DQN) (Mnih et al., 2015), Double DQN (Van Hasselt et al., 2016), Dueling DQN (Wang et al., 2016), Deep SARSA (Rummery & Niranjan, 1994). We also support Bootstrapped DQN (Osband et al., 2016) alongside its corresponding `exploration_module`.

- Actor-critic methods: Soft Actor-Critic (SAC) (Haarnoja et al., 2018), Deep Deterministic Policy Gradient (DDPG) (Silver et al., 2014), Twin-delayed Deep Deterministic Policy Gradient (TD3)

[3]We assume the history H_t also includes the observation O_t. Therefore, the state representation S_t is a function of the history H_t.

[4]In this way, we implement what is typically referred to as "state dependent action space" in the literature.

[5]We use the general term "return" to refer to rewards for bandit settings and Q-values for the MDP setting.

[6]In this iteration, we only support bandit learning algorithms that do not require special neural network architectures. Epistemic Neural Network based contextual bandit algorithms Osband et al. (2023); Zhu & Van Roy (2023); Lu & Van Roy (2017) will be released in the next version of `Pearl`.

```python
    observation_dim = env.observation_space.shape[0]
    agent = PearlAgent(
        policy_learner=DeepQLearning(
            state_dim=observation_dim,
            action_space=env.action_space,
            hidden_dims=[64, 64],
            training_rounds=20,
            exploration_module=EGreedyExploration(epsilon=0.05),
        ),
        history_summarization_module=StackingHistorySummarizationModule(
            observation_dim=observation_dim, history_length=3
        ),
        safety_module=QuantileNetworkMeanVarianceSafetyModule(
            variance_weighting_coefficient=0.1
        ),
        replay_buffer=FIFOOffPolicyReplayBuffer(10_000),
    )
    num_episodes = 500
    for i in range(num_episodes):
        g = 0
        observation, action_space = env.reset()
        agent.reset(observation, action_space)
        done = False
        while not done:
            action = agent.act()
            action_result = env.step(action)
            g += action_result.reward
            agent.observe(action_result)
            done = action_result.done
        agent.learn()
        print(f"Episode {i}: return={g}")
```

(a) `PearlAgent` Episodic Environment Interaction

```yaml
agent:
  _target_: PearlAgent
  policy_learner:
    _target_: DeepQLearning
    state_dim: ${env.observation_dim}
    action_space: ${env.action_space.n}
    hidden_dims:
      - 64
      - 64
    training_rounds: 20
    exploration_module:
      _target_: EGreedyExploration
      epsilon: 0.05

  history_summarization_module:
    _target_: StackingHistorySummarizationModule
    observation_dim: ${env.observation_dim}
    history_length: 3

  safety_module:
    _target_: QuantileNetworkMeanVarianceSafetyModule
    variance_weighting_coefficient: 0.1

  replay_buffer:
    _target_: FIFOOffPolicyReplayBuffer
    capacity: 10000
```

(b) Hydra Configuration for a `PearlAgent`

Figure 2: `PearlAgent` Interaction Interface and Hydra Configeration

(Fujimoto et al., 2018), Proximal Policy Optimization (PPO) (Schulman et al., 2017), and Policy Gradient (REINFORCE) (Sutton et al., 1999).

- Offline methods: Conservative Q-learning (CQL) (Kumar et al., 2020) and Implicit Q-learning (IQL) (Kostrikov et al., 2021).
- Distributional RL: Quantile Regression DQN (QRDQN) (Dabney et al., 2018).

2.1.3 Exploration Module

The `exploration_module` complements policy learners by providing the agent with an *exploration policy*. `Pearl` implements the following set of commonly used exploration modules:

- Random exploration: ϵ-greedy (Sutton & Barto, 2018), Gaussian exploration for continuous action spaces (Lillicrap et al., 2015), and Boltzmann exploration (Cesa-Bianchi et al., 2017).
- Posterior sampling-based exploration: Ensemble sampling (Lu & Van Roy, 2017) and Linear Thompson sampling (Agrawal & Goyal, 2013). Ensemble sampling supports the notion of "deep exploration" proposed by Osband et al. (2016), which enables temporally consistent exploration by acting greedily with respect to an approximate posterior sample of the optimal value function.
- UCB-based exploration: Linear upper confidence bound (LinUCB) (Li et al., 2010) and Neural LinUCB (Xu et al., 2021).

Existing implementations of RL and contextual bandit algorithms, typically implement a policy learner with a fixed exploration strategy (e.g., DQN is usually paired with ϵ-greedy). However, Pearl's modular design opens the door to the possibility of "mixing-and-matching" policy learners with exploration modules. Our hope is that this modular design philosophy this can lead to more performant RL and CB solutions in practice, in addition to helping researchers quickly test new methodological ideas.

2.1.4 Safety Module

The safety module in `Pearl` is currently designed to offer three main features.

- A `risk_sensitive_safety_module`, which facilitates risk sensitive learning with distributional policy learners. Each `risk_sensitive_safety_module` implements a method to compute a value (or Q-value) function from a distribution over value functions under a different risk metric, and can conform to different risk preferences of an RL agent.

- A `filter_action` safety interface allows the agent designer to specify heuristics or environment constraints to only select state-dependent safe action spaces at each step.

- A `reward_constrained_safety_module` which allows the pearl agent to learn in constrained MDPs, with the idea of bounding the long-run costs of a learned policy below a threshold[7]. We use Reward Constraint Policy Optimization (RCPO) (Tessler et al., 2018) in this safety module since it can be applied to different policy optimization algorithms, can work with general cost constraints and is reward agnostic.

2.1.5 History Summarization Module

The `history_summarization_module` implements two key functionalities, keeping track of the history at any environment interaction step and summarizing a history into a state representation.

- During the environment interaction step, the `history_summarization_module` adds (H_{t-1}, H_t) to the agent's replay buffer when the environment is non-Markovian. It also updates the agent's state using the interaction tuple $\mathcal{E}_t$ and history H_{t-1}, which can be used by the `policy_learner` to compute an action at the next time step $t+1$.

- During training, a batch of history transitions $\{(H_{i-1}, H_i)\}$ are sampled from the replay buffer. The `history_summarization_module` computes the corresponding state representations and generates a batch of interaction tuples for the `PearlAgent` to update other modules.

In our current implementation for `PearlAgent`'s `history_summarization_module`, we support both naive history stacking and long-short-term-memory (LSTM) (Hochreiter & Schmidhuber, 1997) based history summarization.

2.1.6 Replay Buffer

The notion of replay buffer, a container for storing previously observed experiences, is central to RL as it enables *experience replay*, the reuse of past experience to improve learning (Lin, 1992). In addition to sub-setting the most informative experiences, replay buffers allow for efficient data reuse by breaking the temporal correlations in sequential data. The `replay_buffer` module in `Pearl` implements several versions of a replay buffer.

- `FIFOOffPolicyReplayBuffer` is based on a first-in-first-out queue and stores interaction tuples for the off-policy setting. For on-policy settings, we provide an extension in the form of `FIFOOnPolicyReplayBuffer`[8].

- `BootstrapReplayBuffer` (Osband et al., 2016) implements *bootstrap masks*. We also build `HindsightExperienceReplayBuffer` with *goal replacement* (Andrychowicz et al., 2017)

2.2 Agent Usage

Figure 2a illustrates a typical episodic environment interaction loop where an agent learns a policy for an environment with Deep Q-learning. Here, learning occurs at the end of each episode. The Pearl `Environment` class is based on the `step` method, which returns an `ActionResult` containing reward, next state, and whether the episode has been truncated, terminated, or done. The `PearlAgent` class accepts optional arguments for components such as history summarization module or safety module

[7]Users can specify both the per-step costs as well as the threshold.

[8]Although replay buffers are not typically used in the on-policy setting, we are able to unify off- and on-policy methods using this abstraction.

Table 1: Comparison of Pearl agent to alternative popular RL libraries

Features	ReAgent	RLLib	SB3	Tianshou	CleanRL	Pearl
Modularity	✗	✗	✗	✗	✗	✓
Intelligent Exploration	✗	✗	✗	✓	✗	✓
Safety	✗	✗	✗	◦[9]	◦[9]	✓
History Summarization	✗	✓	✗	✗	✗	✓
Data Augmented Replay Buffer	✗	✓	✓	✓	✓	✓
Contextual Bandit	✓	◦[10]	✗	✗	✗	✓
Offline RL	✓	✓	✓	✓	✗	✓
Dynamic Action Space	✓	✗	✗	✗	✗	✓

(with no-op components being the default). In our example, we specify a history summarization module that stacks the last three states and a safety module seeking to minimize variance. Likewise, policy learner classes accept an optional exploration module argument; in this example, we use an ϵ-greedy exploration with $\epsilon = 0.05$. In practice, it is more convenient to specify agents and environments via Hydra (Yadan, 2019) configuration files supported by `Pearl`, which provides a convenient way of running experiments and hyperparameter tuning. A Hydra file generating the same agent as above is shown in Figure 2b.

3 Comparison to Existing Libraries

To illustrate the differences between `Pearl` with other existing RL libraries, we compared `Pearl`'s functionalities to four popular RL libraries, namely, ReAgent (), RLLib (Liang et al., 2018), Stable-Baselines3 (Raffin et al., 2021), Tianshou (Weng et al., 2022), and CleanRL (Huang et al., 2022). The main motivation of these libraries is to facilitate reproducible benchmarking of existing RL algorithms.

As highlighted in Table 1, `Pearl` implements several capabilities that are crucial for end-to-end application of an RL system, such as ability to perform structured exploration, offline learning, and safety considerations. Modular design allows users to test performance with different combinations of features. In addition, `Pearl` crucially supports dynamic action spaces, which is an common setting in practical applications. `Pearl` also explicitly supports bandit policy learners along with the corresponding exploration algorithms. Bandit settings find widespread use in large scale industry applications.

We mention a few other RL libraries we surveyed while designing `Pearl`. The d3RLpy (Seno & Imai, 2022) library only provides algorithm implementations for offline and online (specifically, off-policy algorithms) policy learning. Besides, contextual bandit methods are not supported by d3Rlpy. TorchRL (Bou et al., 2023) is a recent modular RL library that implements pytorch and python based primitives which can be used to develop RL systems. Unlike `Pearl` , TorchRL is designed keeping in mind components which are typically used in a policy learning algorithm implementation. Agent design with features like exploration, safe learning etc. is not the focus of TorchRL. Lastly, the Vowpal Wabbit library (Agarwal et al., 2014) offers a rich and diverse set of contextual bandit algorithm implementations, tested on multiple domains and environments. However, to the best of our knowledge, it is designed to exclusively support bandit learning settings and does not explicitly have PyTorch support.

[9]Even though Tianshou and CleanRL have implementations of quantile regression DQN and/or C51, these are more like standalone algorithm implementations which do not implement generic risk sensitive learning. In addition, none of the existing libraries implement policy learning with constraints for different policy optimization algorithms. This is because most existing libraries focus almost entirely on implementing policy learning algorithms without giving considerations to other features.

[10]Only supports linear bandit learning algorithms.

4 Benchmark

4.1 Reinforcement Learning Benchmarks

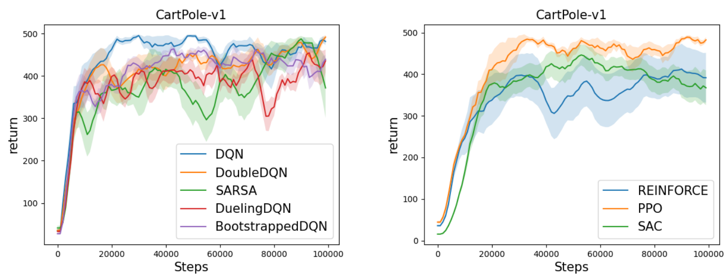


Figure 3: Training returns of discrete control methods on the CartPole task. The left and right panels show returns for value- and policy-based methods, respectively.

We first benchmarked a `PearlAgent` with its discrete control methods on a classic reinforcement learning task called Cartpole. Experiment details are omitted and can be found in our code base. We plotted learning curves of the agent's achieved returns in Figure 3. The x axis shows the number of environment steps while the y axis shows the average of episodic returns received over the past 5000 steps. Each experiment was performed with 5 different random seeds that which fully control the stochasticity the experiments. Shading areas stand for ± 1 standard error across different runs. These results are only meant to serve as a sanity check since reproducible research is only one of Pearl's motivation – we only checked for stable, consistent learning of our implementations rather than searching for the best training runs with optimal hyperparameter choices.

We then benchmarked a `PearlAgent` with three different actor-critic algorithms on continuous control tasks in Mujoco. The results are shown in Figure 4 below. We tested soft-actor critic (SAC), discrete deterministic policy gradients (DDPG) and twin delayed deep deterministic policy gradients (TD3), for a set of commonly used hyperparameters, without tuning them. The axes have the same meaning as those in the discrete control experiments.

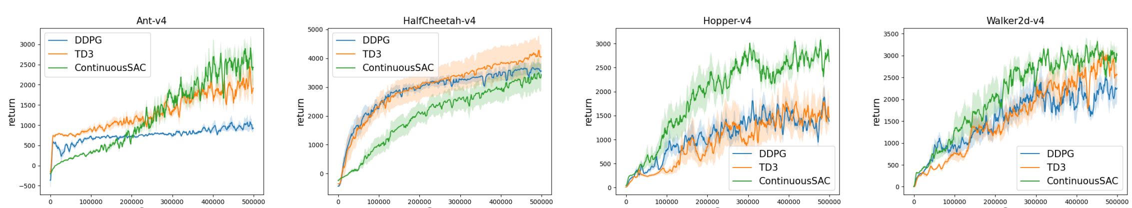


Figure 4: Training returns of SAC, DDPG and TD3 on four Mujoco continuous control tasks.

We also test our offline algorithms, specifically, Implicit Q learning (IQL), on continuous control tasks with offline data. Instead of integrating with D4RL which has dependencies on older versions of Mujoco, we created our own offline datasets following ideas outlined in the D4RL paper (Fu et al., 2020). We create a small dataset of 100k transitions by training a soft-actor critic (SAC) based agent with a high entropy coefficient. The dataset comprises of all transitions in the SAC agent's replay buffer, akin to how the "medium" dataset was generated in the D4RL paper. In table 2 below, we report normalized scores using the same hyperparameters used in the IQL paper (Kostrikov et al., 2021).

We also test our implementation of DQN on Atari games with 3 seeds, with the same convolutional neural network architecture as reported in (Mnih et al., 2015), and achieved reasonable performance in Pong, Beamrider and Breakout in 5 million steps. See 3 for more details.

Environment	Random return	IQL return	Expert return	Normalized score
HalfCheetah-v4	-426.93	145.89	484.80	0.62
Walker2d-v4	-3.88	1225.12	2348.07	0.52
Hopper-v4	109.33	1042.03	3113.23	0.31

Table 2: Normalized scores of Implicit Q learning on different continuous control Mujoco environments. "Random return" refers to the average return of an untrained SAC agent. "IQL return" refers to the average evaluation returns of the trained IQL agent (episodic returns of the trained agent when interacting with the environment). "Expert return" is the maximum episodic return in the offline dataset.

Agent	Breakout	BeamRider	Pong
DQN	151.00 ± 21.82	5351.94 ± 400.50	19.22 ± 0.45

Table 3: Average performance of our DQN implementation on Atari games.

4.2 Neural Contextual Bandits Benchmarks

We implemented and tested the performance of neural adaptations of common CB algorithms. Our benchmarks consists of datasets from the UCI repository (Asuncion & Newman, 2007), adapted to CB interaction model. The results are depicted in Figure 5. Using supervised learning datasets for testing CB algorithms is common in past literature (Dudík et al., 2011; Foster et al., 2018; Bietti et al., 2021). We tested neural implementations of the LinUCB, Thompson Sampling (TS), and SquareCB (Li et al., 2010; Agrawal & Goyal, 2013; Foster & Rakhlin, 2020). This showcases the simplicity of combining deep neural networks within CB algorithms in `Pearl` due to its PyTorch support. See Appendix A for benchmark setup and implementation details.

4.3 Agent Versatility Benchmark

This section provides an initial assessment of `Pearl`'s four primary abilities – summarizing the history to handle partial observable environments, exploring effectively to achieve rewards even when they are sparse, learning with cost constraints, and learning risk-averse policies.

History Summarization for Partial Observability: To test `Pearl`'s ability to handle partial observability, we adapted *Acrobot*, a fully observable, classic reinforcement learning environment, to a partial observable variant. In this environment, the goal is to swing up a chain connected by two linkes. In the original Acrobot environment, the agent can perceive the angles and anglular velocities of the two links. In our partial observable variant, only the angles are observable. Consequently, the agent must use both current and past observations to deduce the angular velocities, which is crucial for selecting the optimal action. To further increase the degree of partial observability, the new environment is designed to emit its observation every 2 steps and to emit an all-zero vector for the rest of time steps.

We tested `Pearl`'s LSTM `history_summarization_module` to see if it can handle the partial observability challenge presented in the above environment. The base algorithm was the DQN algorithm (Mnih et al., 2015). We plotted the mean and the standard error of the achieved returns in Figure 6a. It shows that 1) without the LSTM `history_summarization_module`, the agent did not achieve any progress of learning, 2) with the `history_summarization_module`, the agent achieves a significantly better performance.

Effective Exploration for Sparse Rewards: To test the agent's capability to explore, we implemented the *DeepSea* environment (Osband et al., 2019), known for its exploration challenge. The DeepSea environment has $n \times n$ states and is fully deterministic. Our experiments chose $n = 10$, in which the chance of reaching the target state under a random policy is 2^{-10}. We tested Pearl's implementation of the Bootstrapped DQN algorithm, which is an exploration algorithm introduced by Osband et al. (2016). Again, DQN was used as the baseline. Figure 6b shows the learning curves

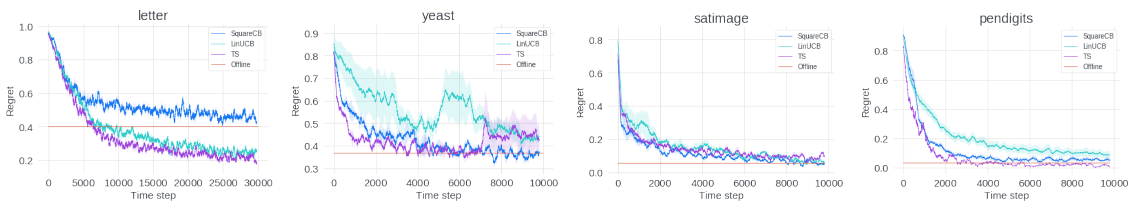
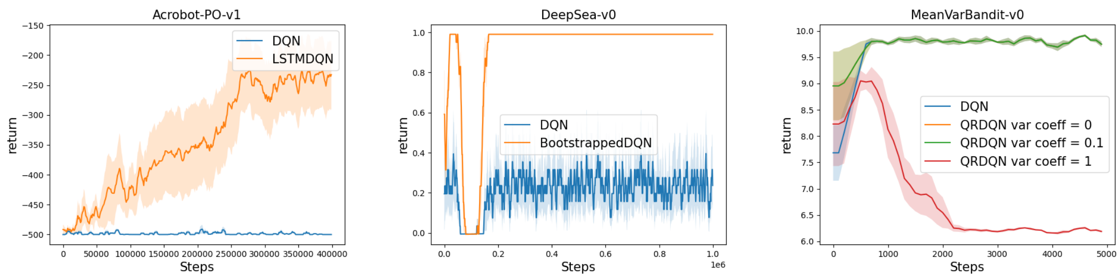


Figure 5: Performance of neural implementations in Pearl of LinUCB, TS and SquareCB on UCI dataset and an offline baseline that is considered near optimal.



(a) (b) (c)

Figure 6: Agent Versatility Benchmark Results: (a) Return of DQN with and without LSTM in the partial-observable Acrobot-v1 environment. (b) DQN and Bootstrapped DQN in a 10×10 Deep Sea environment. (c) One may learn a policy that prefers lower variance return using QRDQN with a large β.

of the two tested algorithms. It can be seen that, Bootstrapped DQN achieved the optimal policy while DQN did not. This suggests that Bootstrapped DQN can perform much better exploration in sparse reward environments.

Learning Risk-Averse Policies: We designed a simple environment called *Stochastic Bandit* to test if `Pearl` can learn policies that fulfills various degrees of safety needs, by balancing the expectation and the variance of the return. StochMDP only has one state and two actions. The reward of each of the two actions follows a Gaussian distribution. The reward distribution for Action 1 has a mean of 6 and a variance of 1. For Action 2, the mean is 10 and the variance is 9. With the classic reinforcement learning formulation, the goal is to maximize the expected return. Therefore the optimal policy is to always choose Action 2. When the agent wants to maximize the mean of the return while minimizing the variance, it chooses a weight scalar β that balances these two terms. Depending on the weight scalar, the optimal policy either always chooses Action 1 or always chooses Action 2. The threshold value for the weight scalar is 0.5 because $6 - 0.5 \times 1 = 10 - 0.5 \times 9$. While this environment is simple, it can serve as a sanity-check to see whether the test algorithm indeed balance mean and variance as predicted.

We tested our implementation of the QR-DQN algorithm (Dabney et al., 2018), which is an algorithm that learns the distribution of the return. Using the learned distribution, the algorithm can estimate the mean and the variance of the return, and further maximize the mean while minimizing the variance. Each experiment has 5 runs, each of which consists of 5000 steps. It can be seen from Figure 6c that, after training, the agent preferred the lower variance action (Action 1) when β was high and preferred the higher variance action (Action 2) when β was low. Our experiment result shows that the algorithm has the ability of learning risk-averse policies.

Learning with Cost Constraints: In many real world problems an agent is required to find an optimal policy subject to cost constraint(s), often formulated as constrained MDPs. For many real world problems where a reward signal is not well defined, it might be useful to specify desirable behavior in the form constraints. For example, limiting the power consumption of a motor can be a

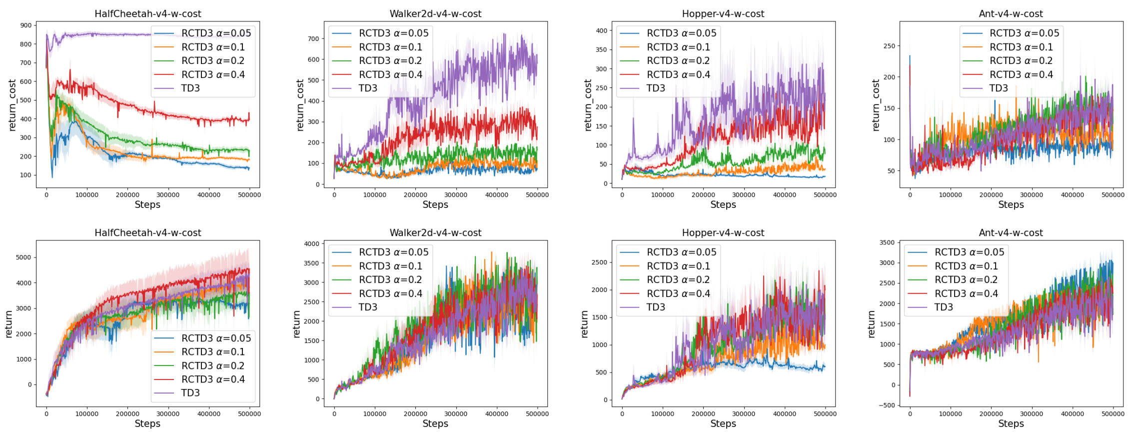


Figure 7: Episodic cost (top) and episodic return (bottom) plots during training on continuous control tasks with cost and reward feedback. The plots present performance of TD3 and our cost constraint adaptation of TD3, RCTD3, for multiple values of constraint threshold α. See text for details.

desirable constraint for learning robotic locomotion. Optimizing for reward subject to constraint(s) requires modification to the learning procedure.

To test policy optimization with the `reward_constrained_safety_module`, we modified a gym environment with a per step cost function, $c(s, a)$, in addition to the standard reward. We choose the per-step cost $c(s, a) = a^2$, which approximates the energy spent in taking action a. Figure 7 shows the results of Reward Constraint TD3 (RCTD3) agent, a `PearlAgent` which uses TD3 as the `policy_learner` along with the `reward_constrained_safety_module`. We chose a normalized cumulative discounted costs as our constraint function with α as the threshold value, namely:

$$(1-\gamma)\,\mathbb{E}_{s\sim\eta_\pi, a\sim\pi}\left[\sum_{t=0}^{\infty}\hat{\gamma}^t c(s_t, a_t)\,|\,s_0 = s, a_0 = a\right] \leq \alpha$$

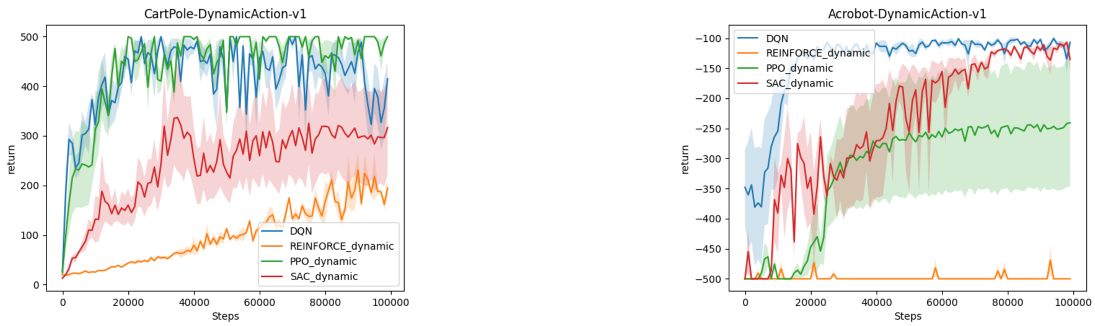
Figure 7 shows cumulative costs decreasing with a smaller value of α for different continuous control Mujoco tasks. Therefore, an RCTD3 agent optimizes for long-run rewards under the cumulative cost constraint as shown above. Interestingly, moderate values of α in different environments does not lead to a significant performance degradation, despite controlling for energy consumption of the control policy.

Adapting to Dynamic Action Spaces In many real-world scenarios, agents must adapt to environments offering varying action spaces at each time step. A quintessential example is recommender systems, where the agent encounters a distinct set of content to recommend to users at each interval. To evaluate Pearl's adaptability to these dynamic action spaces, we crafted two environments based on CartPole and Acrobot. In these environments, every four steps, the agent loses access to action 1 in CartPole and action 2 in Acrobot, respectively.

Figure 8 depicts the learning curves for DQN, SAC, PPO, and REINFORCE within these specialized environments. Despite the increased complexity posed by dynamic action spaces, most agents successfully developed effective policies after 100,000 steps. Notably, REINFORCE consistently underperformed in comparison to other algorithms.

5 Example Industry Product Adoptions

We present three industry product adoptions of `Pearl` as demonstration of `Pearl`'s capability of serving production usage. See Table 4 for how `Pearl` supports these product requirements.



(a) (b)

Figure 8: Dynamic Action Space Benchmark Results: Return of DQN, SAC, PPO and REINFORCE on CartPole and Acrobot environments where each environment deletes an action from the action space every 4 steps. Note that DQN automatically adapts to dynamic action space whereas SAC, PPO and REINFORCE require a special actor neural network.

Table 4: `PearlAgent` Satisfies Requirements of Real-World Applications

`Pearl` Features	Auction RecSys	Ads Auction Bidding	Creative Selection
Policy Learning	✓	✓	✓
Online Exploration		✓	✓
Safety		✓	
History Summarization		✓	
Replay Buffer	✓	✓	✓
Contextual Bandit			✓
Offline RL	✓	✓	
Dynamic Action Space	✓		✓
Large-Scale Neural Network	✓		

Auction-Based Recommender System (Auction RecSys): Optimizing for long-term value in auction-based recommender systems using reinforcement learning presents a significant challenge. This is because it necessitates the integration of a RL agent with a mechanism rooted in supervised learning. In the study by Xu et al. (2023), an on-policy RL solution for auction-based recommendations was introduced, which incorporated `Pearl` during its recent production implementation. Given that the recommender system is heavily influenced by the system's auction policy, the RL agent must undergo on-policy learning offline. This ensures that the newly derived policy, when combined with the auction policy, proveably outperforms the current production system in terms of long-term rewards. As it pertains to recommender systems, a unique set of recommendations is available to users at each step. This necessitates the RL agent's capability to handle a dynamic action space. Additionally, in this implementation, large-scale neural networks were integrated with `Pearl` to offer accurate predictions of value functions for intricate user-recommendation interactions.

Ads Auction Bidding: Real-time bidding in advertising is recognized as a sequential decision-making problem. This is because a bidding agent must efficiently allocate an advertiser-defined budget over a specific duration to maximize the advertiser's conversions. In the study by Korenkevych et al. (2023), the focus is on enhancing the bidding agent's performance amidst the constantly evolving auction market. An RL bidding agent is tasked with prudently learning an offline policy that ensures neither over-expenditure nor under-utilization of the predetermined budget within the set timeframe. Given that the data collected in production is driven by a deterministic policy, the agent needs to engage in limited exploration to gather more insightful data via online exploration. Moreover, due to the inherent volatility of auction markets, which are often only partially observable, the agent is expected to make decisions based on summarized representations of its entire interaction history.

Creative Selection: Beyond sequential decision problems, contextual bandit problems are also prevalent in industry settings. In creative selection for content presentation, where each piece of content has dozens of different available creatives, we adopt a `PearlAgent` with (contextual) neural bandit learner and the Neural LinUCB `exploration_module` for efficient online exploration in learning users' preferences in creatives with minimal number of interactions. Since each content has a different set of available creatives for adoption, the agent is required to support dynamic action space in this problem.

6 Conclusion

The field of RL has witnessed remarkable successes in recent tears, yet, implementation of RL agents in real-world scenarios is still a daunting task. The introduction of `Pearl` marks a significant stride towards bridging this gap, offering a comprehensive, production-ready solution that addresses the multifaceted challenges inherent in RL. By encompassing features like intelligent exploration, safety, history summarization, dynamic action spaces, and support for both online and offline policy optimization, `Pearl` stands out as a versatile tool tailored for diverse real-world applications. We believe that `Pearl` will serve as a valuable resource for the broader adoption of RL in real-world applications, fostering innovation and expanding the boundaries of the field.

References

Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *The Journal of Machine Learning Research*, 15(1):1111–1133, 2014.

Shipra Agrawal and Navin Goyal. Thompson sampling for contextual bandits with linear payoffs. In *International conference on machine learning*, pp. 127–135. PMLR, 2013.

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

Arthur Asuncion and David Newman. Uci machine learning repository, 2007.

Alberto Bietti, Alekh Agarwal, and John Langford. A contextual bandit bake-off. *The Journal of Machine Learning Research*, 22(1):5928–5976, 2021.

Albert Bou, Matteo Bettini, Sebastian Dittert, Vikash Kumar, Shagun Sodhani, Xiaomeng Yang, Gianni De Fabritiis, and Vincent Moens. Torchrl: A data-driven decision-making library for pytorch, 2023.

Nicolò Cesa-Bianchi, Claudio Gentile, Gábor Lugosi, and Gergely Neu. Boltzmann exploration done right. *Advances in neural information processing systems*, 30, 2017.

Will Dabney, Mark Rowland, Marc Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

Miroslav Dudík, John Langford, and Lihong Li. Doubly robust policy evaluation and learning. *arXiv preprint arXiv:1103.4601*, 2011.

Dylan Foster and Alexander Rakhlin. Beyond ucb: Optimal and efficient contextual bandits with regression oracles. In *International Conference on Machine Learning*, pp. 3199–3210. PMLR, 2020.

Dylan Foster, Alekh Agarwal, Miroslav Dudík, Haipeng Luo, and Robert Schapire. Practical contextual bandits with regression oracles. In *International Conference on Machine Learning*, pp. 1539–1548. PMLR, 2018.

Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. D4rl: Datasets for deep data-driven reinforcement learning. *arXiv preprint arXiv:2004.07219*, 2020.

Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pp. 1587–1596. PMLR, 2018.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pp. 1861–1870. PMLR, 2018.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022. URL http://jmlr.org/papers/v23/21-1342.html.

Dmytro Korenkevych, Frank Cheng, Artsiom Balakir, Alex Nikulkov, Lingnan Gao, Zhihao Cen, Zuobing Xu, and Zheqing Zhu. Offline reinforcement learning for optimizing production bidding policies. *unpublished manuscript*, 2023.

Ilya Kostrikov, Ashvin Nair, and Sergey Levine. Offline reinforcement learning with implicit q-learning. In *International Conference on Learning Representations*, 2021.

Aviral Kumar, Aurick Zhou, George Tucker, and Sergey Levine. Conservative q-learning for offline reinforcement learning. *Advances in Neural Information Processing Systems*, 33:1179–1191, 2020.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pp. 661–670, 2010.

Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2018.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8:293–321, 1992.

Xiuyuan Lu and Benjamin Van Roy. Ensemble sampling. *Advances in neural information processing systems*, 30, 2017.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped dqn. *Advances in neural information processing systems*, 29, 2016.

Ian Osband, Benjamin Van Roy, Daniel J Russo, Zheng Wen, et al. Deep exploration via randomized value functions. *J. Mach. Learn. Res.*, 20(124):1–62, 2019.

Ian Osband, Zheng Wen, Seyed Mohammad Asghari, Vikranth Dwaracherla, Morteza Ibrahimi, Xiuyuan Lu, and Benjamin Van Roy. Approximate thompson sampling via epistemic neural networks. *arXiv preprint arXiv:2302.09205*, 2023.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35: 27730–27744, 2022.

Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE international conference on robotics and automation (ICRA)*, pp. 3803–3810. IEEE, 2018.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html.

Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Takuma Seno and Michita Imai. d3rlpy: An offline deep reinforcement learning library. *Journal of Machine Learning Research*, 23(315):1–20, 2022. URL http://jmlr.org/papers/v23/22-0017.html.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pp. 387–395. Pmlr, 2014.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

Chen Tessler, Daniel J Mankowitz, and Shie Mannor. Reward constrained policy optimization. *arXiv preprint arXiv:1805.11074*, 2018.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pp. 1995–2003. PMLR, 2016.

Jiayi Weng, Huayu Chen, Dong Yan, Kaichao You, Alexis Duburcq, Minghao Zhang, Yi Su, Hang Su, and Jun Zhu. Tianshou: A highly modularized deep reinforcement learning library. *Journal of Machine Learning Research*, 23(267):1–6, 2022. URL http://jmlr.org/papers/v23/21-1127.html.

Pan Xu, Zheng Wen, Handong Zhao, and Quanquan Gu. Neural contextual bandits with deep representation and shallow exploration. In *International Conference on Learning Representations*, 2021.

Ruiyang Xu, Jalaj Bhandari, Dmytro Korenkevych, Fan Liu, Yuchen He, Alex Nikulkov, and Zheqing Zhu. Optimizing long-term value for auction-based recommender systems via on-policy reinforcement learning. *RecSys*, 2023.

Omry Yadan. Hydra - a framework for elegantly configuring complex applications. Github, 2019. URL https://github.com/facebookresearch/hydra.

Zheqing Zhu and Benjamin Van Roy. Scalable neural contextual bandit for recommender systems. In *32nd ACM International Conference on Information and Knowledge Management (CIKM)*, pp. 3636–3646, 2023.

A Details on CB implementation

The CB benchmark environment is designed as follows. We assume access to an offline dataset $\{(x_i, y_i)\}_i$, where for every i, $x_i \in \mathbb{R}^d$ is a feature vector, and $y_i \in \mathcal{Y}$ is a label from a finite alphabet. At each time step an agent observes a feature vector x_t and is required to choose an action, $a_t \in \mathcal{Y}$, which is an element of the alphabet of possible labels. The reward model is $r_t = 1\{a_t = y_t\} + \xi$ where $\xi \sim \mathcal{N}(0, \sigma_\xi)$. This type of environments has an explicit exploration challenge: if an agent does not explore correctly it may never receive information on the correct label.

We used a two-layer neural architecture as the reward function approximation in all algorithms we experiment with. The reward function network receives as an input the feature vector and an action (x, a), and returns real value number. The reward model was optimized via PyTorch, by iteratively taking gradients on the standard MSE loss using an Adam optimizer. Beside of the neural versions of LinUCB, TS, and SquareCB we implemented an additional baseline offline approach. For the offline baseline an agent gathers data with a fixed exploratory behavior policy. Then we trained a reward model on this offline dataset and tested its online performance by following the greedy policy with respect to the learned reward model. The architecture of the network and other implementations details of the offline baseline are the same as for the CB algorithms.

General implementation details. Table 5 depicts implementation details corresponding to all benchmarks and algorithms. For the `letter`, `satimage`, `pendigits` that are of higher dimension we used an MLP architecture of $[64, 16]$ hidden layers. For the `yeast` dataset we chose an architecture of $[32, 16]$ hidden layers.

Since the action space of some of these datasets is not small, we chose a binary encoding to the actions. This binary encoding was concatenated to the feature vector. Hence, the input vector of the network has the form (x, a) where a is a binary representation of an element in $|\mathcal{Y}|$, where $\mathcal{Y}$ is the alphabet of possible labels.

The behavior policy with which we gathered data for the offline benchmark was chosen as follows: the behavior policy chooses with probability $1/4$ the correct label, and with probability $3/4$ any label. That allowed to create a balanced dataset, in which the ratio between choosing the correct and incorrect label is small.

The plots presented in Table 1 represent the average performance across 5 runs. The confidence intervals represent the standard error.

LinUCB and TS implementation. Our neural adaptions of LinUCB and TS are based on calculating a bonus term $\|\phi_t(x, a)\|_{A_t^{-1/2}}$ where $\phi_t(x, a)$ is the last layer feature representation of of (x, a) and $A_t = I + \sum_{n=1}^{t-1} \phi_t(x, a)\phi_t(x, a)^T$. For LinUCB we explicitly add this term (Li et al., 2010) after scaling by 0.25, which improved the performance. For the neural version of TS we sampled a reward from a Gaussian distribution with the variance term $\phi_t(x, a)\|_{A_t^{-1/2}}$ and expected reward as calculated by the neural reward model.

Table 5: Implementation Details of the CB algorithms

Architecture	2 layer MLP
Optimizer	Adam
Learning rate	0.01
Batch size	128
Buffer sizer	# of time steps
Action encoding	Binary encoding
Reward variance	$\sigma_\xi = 0.05$

SquareCB implementation. We followed the exact same action decision rule of SquareCB (Foster & Rakhlin, 2020). We chose the scaling parameter $\gamma = 10\sqrt{dT}$ where d is the dimension of the feature-action input vector. We note that setting $\gamma \propto \sqrt{dT}$ was indicated in Foster & Rakhlin (2020) for the linear Bandit problem. We scaled this value by 10 since we empirically observed of an improved performance in our ablation study.