Choice-Based Learning in JAX

Shangyin Tan[*†] **Dan Zheng**[*] **Gordon Plotkin** **Ningning Xie**
Google DeepMind
`{shangyin,danielzheng,plotkin,ningningxie}@google.com`

Abstract

Choice-based learning is a programming paradigm for expressing learning systems in terms of choices and losses. We explore a practical and modular implementation of choice-based learning in JAX by combining two techniques in a novel way: algebraic effects and loss effect operations. We describe the design and implementation of our library, explore its usefulness for real-world applications like hyperparameter tuning and deep reinforcement learning, and compare it with existing approaches.

1 Introduction

Machine learning has achieved many successes in recent decades, but modular and flexible machine learning programming remains a challenge.

Choice-based learning [1, 3, 4] is a recently introduced programming paradigm that can address this challenge. In choice-based learning, programs make *choices* and learn from *feedback* in the form of losses and rewards. This paradigm prioritizes modularity and can be used to implement widely used decision-making techniques, including Markov decision processes and gradient descent.

In this paper, we present our work-in-progress efforts in implementing CHOIX, a library for choice-based learning in Python using JAX. We describe the design and implementation of CHOIX and use it to explore the usefulness of choice-based learning as a modular programming paradigm for machine learning applications.

2 Background

In this section, we explain what choice-based learning is and two programming paradigms that can be used to implement it: algebraic effects and handlers, and losses and choice-loss continuations.

Choice-based learning. Choice-based learning systems make choices based on losses and rewards, utilizing two types of operations: `choose` and `loss`. A `choose` operation takes an input space of choices and and returns an optimal choice based on losses. Then, a `loss`(x) operation records numerical loss values, and accumulated loss values are used by decision strategies to make cost-driven choices.

Algebraic effects and effect handlers. Algebraic effects are a well-explored technique for structured control flow abstraction in programming languages [6, 7]. Programs use abstract effect operations, and effect handlers provide meaning for effect operations.

Algebraic effects are natural for modeling choice-based learning programs: `choose` operations can be represented as effect operations and choice strategies as effect handlers. This allows programmers

[*]Equal contribution.
[†]Work done during internship at Google DeepMind.

Preprint. Under review.

to describe learning algorithms in terms of choices and costs, while leaving choice strategies up to separately-defined optimization algorithms.

Parameterized handlers [7] extend effect handlers by allowing state to be shared between effect operations. This is useful for implementing choice strategies that need to access and update state in a modular way.

Losses and choice-loss continuations. Prior work [1] establishes semantics for choice-based learning languages with `loss` operations. In the semantics, `loss` is a built-in effect operation that can be used freely within programs, including inside handlers. Programs containing `loss` operations are transformed to return an accumulated loss value in addition to their original results.

Additionally, effect handlers can access loss values via a *choice-loss continuation* (or just "loss continuation", `lk`). Loss continuations compute the accumulated loss from using a particular choice; effect handlers can make optimization choices using loss continuations by optionally transforming them (e.g., via automatic differentiation) and calling them with different arguments before resuming the program with the optimal choice.

3 CHOIX: a library for choice-based learning

In this section, we introduce CHOIX[1], a library for choice-based learning in Python using JAX [2]. CHOIX support choice-based learning via the following features: user-defined effect operations and effect handlers (as described in Section 2), `loss` operations, and loss continuations.

In CHOIX, effect operations are declared as Python functions decorated with `@effect`. these functions define an abstract interface and need no implementation. Effect handler functions provide concrete implementations for effect operations. CHOIX provides a `handle` function for effect-handling: `handle` takes a list of handlers and an "effectful" function containing effect operations like `choose` and `loss`, and returns an effect-handled version that returns an accumulated loss. With `handle`, users can write programs using rich features for choice-based learning, then convert them to standard JAX functions to be used normally with other JAX code, including transformations and compilation.

3.1 Examples

In this section, we walk through some machine learning programs written using CHOIX to highlight the benefits of programming in the choice-based learning paradigm.

3.1.1 Linear regression

A simple application of choice-based learning is linear regression: given training data X, Y and initial parameters θ, choose new parameters θ' that minimize a loss function via gradient descent. Figure 1 shows examples of linear regression written in standard JAX and in CHOIX.

With CHOIX, gradient descent is modeled as a `choose` effect that updates parameters: $\theta' = \text{choose}(\theta)$. A loss value l is computed from the updated parameters by comparing predictions with target data, and `loss`(l) is called to provide feedback. A `choose_grad` handler function implements parameter update: it first gets gradients g by differentiating the loss continuation lk with respect to θ, then resumes the program with updated parameters $\theta' = \theta - \eta g$, where η is a fixed learning rate.

Note that we are not presenting the benefit of CHOIX in the linear regression example. Instead, the goal is to demonstrate how to create choice-based learning programs using CHOIX. As the complexity of these choice-based learning examples evolves (Appendix C, Section 3.1.2, Section 3.1.3), we intend to highlight the modularity provided by CHOIX, showcasing its ability to handle more complex choice-based learning applications in a structured manner. For example, Appendix C shows how to extend linear regression with hyperparameter tuning in CHOIX using nested effect operations and `handle`.

[1]"Choix" means "choice" in French.

```
1  def loss_fn(params, x, y):
2      w, b = params
3      prediction = w * x + b
4      return jnp.mean(prediction - y ** 2)
5
6  def update(params, grads, lr):
7      return jax.tree_map(
8          lambda p, g: p - lr * g, params, grads
9      )
10
11 def linear_regression(lr, params, x, y):
12     """Performs one step of linear regression."""
13     loss_value = jax.grad(loss_fn)(params, x, y)
14     grads = grad_fn(params, x, y)
15     new_params = update(params, grads, lr)
16     return new_params
17
18 # Usage.
19 x, y, w, b = ...
20 new_params = linear_regression(
21     lr=0.001, (w, b), x, y
22 )
```

```
1  @effect
2  def choose[T](x: T) -> T:
3    """Choice effect operation."""
4
5  def update(params, grads, lr):
6      return jax.tree_map(
7          lambda p, g: p - lr * g, params, grads
8      )
9
10 def choose_grad_handler(lr, params, k, lk):
11     """Gradient-descent handler."""
12     grads = jax.grad(lk, argnums=(0))(params)
13     new_params = update(params, grads, lr)
14     return k(new_params)
15
16 def linear_regression(params, x, y):
17     """Performs one step of linear regression."""
18     w_new, b_new = choose(params)
19     prediction = w_new * x + b_new
20     loss(jnp.mean((prediction - y) ** 2))
21     return (w_new, b_new)
22
23 # Usage.
24 linear_regression = handle(
25     Handler(parameterized=True, choose=choose_grad_handler)
26 )(linear_regression)(lr=0.001)
27
28 x, y, w, b = ...
29 loss_value, new_params = linear_regression((w, b), x, y)
```

```
  losses: loss-related operations
 effects: effect operations
handlers: effect handlers
```

Figure 1: Linear regression via gradient descent in standard JAX (left) versus choice-based learning in CHOIX (right). In `choose_grad_handler`, the continuation `k` resumes the program from where the effect operation is called (line 18).

3.1.2 Reinforcement learning

Reinforcement learning (RL) is a particularly good fit for choice-based learning, as effect operations provide a modular interface for RL algorithms, and parameterized effect handlers facilitate seamless state sharing (e.g. agent parameters like Q-tables or neural network weights) between operations.

In Appendix D, we show a generic function for RL in CHOIX called `run_episode`. `run_episode` uses multiple effect operations: in particular, agent operations `predict` and `feedback` and an environment operation `observe`. `predict` selects an action based on the current state and agent parameters; `observe` applies the action to the state to get a new state and reward; finally, `feedback` updates agent parameters based on reward and accumulated loss. By implementing handler functions for these operations, we can get a specialized version of `run_episode` for a particular agent algorithm and environment.

Handler functions in Appendix D implement Q-learning [11], a classic RL algorithm that uses a Q-table mappign state-action pairs to expected rewards. The `predict` handler selects an action based on the Q-table and state, and the `feedback` handler updates the Q-table based on the selected action and reward. The `observe` handler here implements environment update for the cliff-walking task [10].

3.1.3 Deep reinforcement learning

A particularly compelling variant of reinforcement learning is deep reinforcement learning, where the RL agent is a deep neural network and the training process aims to update parameters to enhance predictions. Implementing deep reinforcement learning with CHOIX is as simple as reusing the generic RL algorithm from Section 3.1.2 with alternative handlers.

In Appendix E, we present an implementation of proximal policy optimization (PPO) [8], a well-known algorithm in deep reinforcement learning. Compared with the Q-learning example above, the main changes are the `feedback_handler`. The PPO algorithm updates agent network weights utilizing three losses: actor loss, value loss, and entropy. These loss values can naturally be accumulated via `loss` without the need of writing an explicit loss function.

To choose new parameters, we use gradient descent with the Adam optimizer from Optax[2]. This is done by adapting the `choose` effect from the linear regression example and modifying

[2]https://github.com/google-deepmind/optax

the gradient descent handler to use Optax. Notably, `choose_optax_handler` is generic like `choose_grad_handler`: both can be used in other programs for gradient optimization.

Additionally, minor adjustments to the `observe` and `predict` handlers are required, reflecting changes in the agent parameter and environment. In `ppo_predict_handler`, the action prediction is calculated through the deep nerual network. In order to train on Atari games, we change the `observe` handler to `atari_observe_handler` to correctly update the Atari environment.

3.1.4 Summary

The examples above demonstrate the modularity of Choix for writing choice-based learning programs. The separation between abstract effect operations and concrete handler implementations makes the process of developing new choice-based learning programs easier: new programs define their algorithms as abstract effect operations where effect handlers can be shared and composed between programs. For example, most choice-based learning programs that use Optax can reuse `choose_optax_handler` without changing their own application code.

3.2 Implementation

JAX[2] is an open-source system for machine learning research. It offers the familiar API of NumPy with the speed of hardware accelerators via the XLA compiler. JAX's design centers around composable function transformations like automatic differentiation, compilation, and parallelization, making it a solid basis for our work.

CHOIX is implemented as a function transformation on JAXPRS[3], the core language underlying JAX. The transformation is done in multiple stages, producing an *effect-handled* JAXPR at the end. We outline the most important stages, with details in Appendix A:

- **Source program.** Source Python programs using CHOIX are lowered to JAXPRS that contain multiple constructs not present in standard JAXPRS: effect operations, effect handlers, and `loss` operations. All of these must be translated away before CHOIX programs can be executed. (Figure 2b.)
- **Delimited, concrete handlers.** Effect handlers now have explicitly delimited scopes, which is needed to create properly-scoped continuations during effect handling. Handler functions are transformed from Python functions to JAXPRS via tracing. (Figure 2c.)
- **Loss transformation.** Performs the selection monad transformation [1]: `loss` operations and choice-loss continuations are translated away, and translated programs compute and return accumulated loss values. (Figure 3a.)
- **Effect handling.** Replaces effect operations with their handler implementations, reifying delimited program continuations based on handler scope. Effect operations and effect handlers are translated away. (Figure 3b.)

Finally, we can obtain CHOIX-transformed JAXPRS as in Figure 3c. These JAXPRS do not contain effect operations, effect handlers, or `loss` operations: they work naturally with JAX transformations (e.g. `jax.grad` and `jax.jit`), allowing them to interoperate with other JAX code. CHOIX lets users enjoy the benefits of writing choice-based learning programs using algebraic effects without sacrificing performance.

Optimized runtime performance is not a current focus of CHOIX: since CHOIX produces JAXPRS similar to standard JAX code, we expect similar performance during runtime. In practice, we noticed a slowdown of approximately 0-10% for CHOIX on the example programs included in this paper.

4 Related work

Choice-based learning. Choice-based learning as a programming paradigm has gained attention in recent years[1, 4]. Frameworks like SmartChoices[3] and PyGlove[5] provide abstractions for writing programs in terms of decisions and rewards, but typically provide a fixed search interface

[3]`https://jax.readthedocs.io/en/latest/jaxpr.html`

and limited search algorithms. CHOIX supports flexible search thanks to effect handlers: users can define custom search operations, implement or import search algorithms from external frameworks like Vizier[9], and compose them together to create modular choice-based programs.

Effect handling in JAX. To our knowledge, our work is the first to propose a library for algebraic effects in JAX. Our work is inspired by a simple proof-of-concept[4] of effect handling in JAX goes beyond it to implement a fully-fledged effects system. Oryx[5] is a library for probabilistic programming built on JAX. Oryx provides transformations for writing stateful computations via collecting and injecting tagged values, which can be seen as a limited effect system particularly useful for writing deep learning and probabilistic programs. In CHOIX, state management is done in a similar way via parameterized handlers: we provide a short explanation in Appendix B.

5 Conclusion

Choice-based learning is a programming paradigm in which systems make feedback-guided choices. We introduce CHOIX, a library for choice-based learning in JAX. CHOIX uses (1) algebraic effects and handlers as a modular interface for writing choice-based learning programs, with (2) the selection monad to enable loss-based choice optimization.

We explore several machine learning applications of choice-based learning in CHOIX. We believe it is a promising approach for developing learning programs, including systems applications that replace hardcoded heuristics with learned strategies. We are excited to explore more applications of this programming model and welcome feedback and ideas.

References

[1] Martin Abadi and Gordon Plotkin. "Smart choices and the selection monad". In: *Logical Methods in Computer Science* 19 (2023).

[2] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.4.17. 2018. URL: `http://github.com/google/jax`.

[3] Daniel Golovin et al. *SmartChoices: Augmenting Software with Learned Implementations*. 2023. arXiv: `2304.13033 [cs.SE]`.

[4] Ugo Dal Lago, Francesco Gavazzo, and Alexis Ghyselen. *On Reinforcement Learning, Effect Handlers, and the State Monad*. 2022. arXiv: `2203.15426 [cs.PL]`.

[5] Daiyi Peng et al. *PyGlove: Symbolic Programming for Automated Machine Learning*. 2021. arXiv: `2101.08809 [cs.LG]`.

[6] Gordon D. Plotkin and John Power. "Adequacy for Algebraic Effects". In: *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*. Ed. by Furio Honsell and Marino Miculan. Vol. 2030. Lecture Notes in Computer Science. Springer, 2001, pp. 1–24. DOI: `10.1007/3-540-45315-6\_1`. URL: `https://doi.org/10.1007/3-540-45315-6%5C_1`.

[7] Gordon D. Plotkin and Matija Pretnar. "Handlers of Algebraic Effects". In: *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Giuseppe Castagna. Vol. 5502. Lecture Notes in Computer Science. Springer, 2009, pp. 80–94. DOI: `10.1007/978-3-642-00590-9\_7`. URL: `https://doi.org/10.1007/978-3-642-00590-9%5C_7`.

[8] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: `1707.06347 [cs.LG]`.

[9] Xingyou Song et al. *Open Source Vizier: Distributed Infrastructure and API for Reliable and Flexible Blackbox Optimization*. 2023. arXiv: `2207.13676 [cs.LG]`.

[10] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[4]`https://colab.sandbox.google.com/drive/1HGs59anVC2AOsmt7C4v8yD6v8gZSJGm6`, linked from `http://blog.sigfpe.com`.

[5]`https://github.com/jax-ml/oryx`

[11] Christopher J. C. H. Watkins and Peter Dayan. “Technical Note Q-Learning”. In: *Mach. Learn.* 8 (1992), pp. 279–292. DOI: `10.1007/BF00992698`. URL: `https://doi.org/10.1007/BF00992698`.

Appendix

A CHOIX transformation details

This section walks through the CHOIX program transformation for a toy program, highlighting how it works stage-by-stage.

The toy program is intentionally simple to save space, since transformation outputs are long. Later appendix sections show more sophisticated programs using CHOIX.

```python
@effect
def increment(x: int) -> int:
    pass

def increment_handler(x, k, lk):
    return k(x + 1)

@handle(Handler(increment=increment_handler))
def main(x):
    y = increment(x)
    loss(y ** 2)
    return y

print(main(10))
```

(a) Source program.

```
# Source Jaxpr.
{ lambda ; a:i32[]. let
    _:bool[] = handler[
      handler_impl={'increment': <function increment_handler>}
      in_tree=PyTreeDef(())
      name=increment
      parameterized=False
    ]
    b:i32[] = increment[in_tree=PyTreeDef(*) is_effect_op=True name=increment] a
    c:i32[] = integer_pow[y=2] b
    d:f32[] = convert_element_type c
    _:f32[] = loss d
    _:bool[] = handler_end_marker[name=increment]
  in (b,) }
```

(b) Source program as a JAXPR.

```
# Delimited concrete handler Jaxpr.
{ lambda ; a:i32[]. let
    b:i32[] = delimited_handler[
      args=[]
      body_jaxpr={ lambda ; c:i32[]. let
          d:i32[] = increment[in_tree=PyTreeDef(*) is_effect_op=True name=increment] c
          e:i32[] = integer_pow[y=2] d
          f:f32[] = convert_element_type[new_dtype=float32 weak_type=False] e
          g:f32[] = loss f
        in (d,) }
      handler_impl=[
        increment={ lambda ; h:i32[]. let
            i:i32[] = add h 1
            j:i32[] = call_k_with_loss[name=increment] i
          in (j,) }
      ]
      in_tree=PyTreeDef(())
      name=increment
      parameterized=False
    ] a
  in (b,) }
```

(c) Delimited concrete handler JAXPR.

Figure 2: Initial CHOIX program transformation stages. A Python source program using CHOIX is lowered to a JAXPR. Then, effect handler scopes are made explicit and handler functions are converted from Python to JAXPRS.

```
# Loss-translated Jaxpr.
{ lambda ; a:i32[]. let
    b_loss:f32[] c:i32[] = delimited_handler[
      args=[]
      body_jaxpr={ lambda ; d:i32[]. let
          e:i32[] = increment[in_tree=PyTreeDef(*) is_effect_op=True name=increment] d
          f:i32[] = integer_pow[y=2] e
          g:f32[] = convert_element_type[new_dtype=float32 weak_type=False] f
        in (g, e) }
      handler_impl=[
        increment={ lambda ; h:i32[]. let
            i:i32[] = add h 1
            j_loss:f32[] k:i32[] = call_k_with_loss[name=increment] i
          in (j_loss, k) }
      ]
      in_tree=PyTreeDef(())
      name=increment
      parameterized=False
    ] a
  in (b_loss, c) }
```

(a) Loss-translated JAXPR. Effect operations and handlers now return loss values, accumulated from `loss` calls.

```
# Effect-handling Jaxpr.
{ lambda ; a:i32[]. let
    b_loss:f32[] c:i32[] = closed_call[
      call_jaxpr={ lambda ; d:i32[]. let
          h':f32[] i':i32[] = closed_call[
            call_jaxpr={ lambda ; j:i32[]. let
                k:i32[] = add j 1
                l_loss:f32[] m:i32[] = closed_call[
                  call_jaxpr={ lambda ; n:i32[]. let
                      f:i32[] = integer_pow[y=2] n
                      g:f32[] = convert_element_type[
                        new_dtype=float32
                        weak_type=False
                      ] f
                    in (g, n) }
                ] k
              in (l_loss, m) }
          ] a
        in (h', i') }
    ] a
  in (b_loss, c) }
```

(b) Effect-handled JAXPR. All special constructs from CHOIX (including effect operations and handlers) have been translated away.

```
# Inlined Jaxpr.
{ lambda ; a:i32[]. let
    k:i32[] = add a 1
    f:i32[] = integer_pow[y=2] k
    g:f32[] = convert_element_type[new_dtype=float32 weak_type=False] f
  in (g, k) }
```

(c) Inlined, canonicalized JAXPR. This is a standard JAXPR, ready for execution or transformation.

Figure 3: Final CHOIX program transformation stages, performing the loss translation and effect handling. In the final stage, all choice-based learning constructs are translated away, producing a lowered JAXPR that matches JAXPRs from direct-style JAX code.

B Stateful computation

```python
State = int

@effect
def get() -> State:
    """Gets the state value."""

@effect
def set(s: State) -> None:
    """Updates the state value."""

def program() -> State:
    x1 = get()
    set(x * 2)
    x2 = get()
    set(x2 + 5)
    return x1 + x2 + get()

def get_handler(s: State, x: None, k, lk):
    jax.debug.print('get: s = {}', s)
    return k(s, s)

def set_handler(s: State, x: State, k, lk):
    jax.debug.print('set: s = {}, x = {}', s, x)
    return k(x, None)

x = 42
main = handle(
    Handler(parameterized=True, get=get_handler, set=set_handler)
)(program)(x)

result = main()
# get: s = 42
# set: s = 42, x = 84
# get: s = 84
# set: s = 84, x = 89
# get: s = 89
print(result)
# (loss=0.0, value=Array(215, dtype=int32))

print(jax.make_jaxpr(main)())
# { lambda ; a:i32[]. let
#     b:i32[] = mul a 2
#     c:i32[] = add b 5
#     d:i32[] = add a b
#     e:i32[] = add d c
#   in (0.0, e) }
```

Figure 4: Stateful computation example using CHOIX. This is noteworthy since state management is a primary concern of deep learning libraries built on JAX, as JAX's purely functional programming model does not provide native support for state management[6]. As shown, the final transformed JAXPR is as efficient as direct-style JAX code.

[6]https://jax.readthedocs.io/en/latest/jax-101/07-state.html#taking-it-further

C Hyperparameter optimization

In machine learning, *hyperparameters* are variables (like learning rate) that are not directly related to the training data but govern the training process itself. Hyperparameter values can greatly affect the training process, including how quickly training converges and whether training succeeds at all. *Hyperparameter optimization* (or tuning) is the process of searching for optimal hyperparameters.

Standard machine learning approaches tend to view hyperparameter tuning as a separate and orthogonal process to model training, often using external blackbox optimization frameworks like Vizier [9] that involve nontrivial configuration. This leads to a barrier between training and tuning, making it difficult to extend programs for training to also do hyperparameter search without rewriting code.

With CHOIX, hyperparameter tuning is naturally represented using *nested effects and handlers*, as shown in Figure 5. We define an effect `choose_single` for selecting a hyperparameter configuration from a search space. This lets us write a `hyperparameter_tuning` generic function that takes a search space and applies `choose_single` to select hyperparameters for an argument training function `f`.

Hyperparameter optimization approaches can then be implemented as handlers, too. Here, the `choose_enumerate_handler` function defines a simple grid search that exhaustively explores a search space and picks the option minimizing loss. Other handlers could implement more advanced features like random exploration or early-stopping. Existing libraries for hyperparameter search like Vizier [9] could also be used to implement handlers.

```
@effect
def choose_single[T](x: Sequence[T]) -> T:
  """Choose-one-from-many effect operation."""

def hyperparameter_tuning[T](hparam_options: Sequence[T], f: Callable[[T], Any]):
    hparam = choose_single(hparam_options)
    return hparam, f(hparam)

# Handlers.
def choose_enumerate_handler[T](options: Sequence[T], k, lk):
    """Enumerative choice handler, using a vectorized loss function."""
    losses = jax.vmap(lk)(jnp.asarray(options))
    best_option = options[jnp.argmin(losses)]
    return k(best_option)

# Usage.
hyperparameter_tuning = handle(
    Handler(choose_single=choose_enumerate_handler),
)(hyperparameter_tuning)

lrs = [0.001, 0.002, 0.005, ...]
params, x, y = ...

def f(lr: float):
    g = handle(
        Handler(parameterized=True, choose=choose_grad_handler),
    )(linear_regression)
    return g(lr)(params, x, y)

loss_value, (optimal_hparam, new_params) = hyperparameter_tuning(lrs, f)
```

Figure 5: Hyperparameter tuning with CHOIX.

D Reinforcement learning

```python
# Effect operations for reinforcement learning.
@effect
def predict(state: State) -> Action:
    """Selects an action given an environment state."""

@effect
def observe(state: State, action: Action) -> tuple[Reward, State]:
    """Performs an action to get a reward and new environment state."""

Feedback = tuple[State, Action, Reward, State] # (old_state, action, reward, new_state)

@effect
def feedback(data: Feedback) -> None:
    """Updates agent with reward feedback."""

# Generic reinforcement learning algorithm.
def run_episode(state: State, is_goal: Callable[[State], bool]) -> State:
    """Runs reinforcement learning for one episode, updating environment with actions from agent."""
    def cond(state: State):
        return jnp.invert(is_goal(state))

    def body(state: State):
        action = predict(state)
        reward, new_state = observe(state, action)
        feedback((state, action, reward, new_state))
        return new_state

    return lax.while_loop(cond, body, state)

# Random number generation handler.
def random_uniform_handler(key: PRNGKey, args: tuple[int, int], k, lk):
    minval, maxval = args
    key, subkey = random.split(key)
    return k(key, random.uniform(subkey, minval, maxval))

# Agent handlers: Q-learning.
def predict_handler(params: Theta, state: State, k, lk):
    # Selects an action given agent parameters and state.
    epsilon = random_uniform(0, 1)
    action = jax.lax.select(
        epsilon < 0.1,
        jnp.floor(random_uniform(0, action_count)),
        find_max_action(params, state)
    )
    return k(params, action)

def feedback_handler(params: Theta, data: Feedback, k, lk):
    # Updates agent parameters given feedback.
    max_reward = find_max_reward(params, data)
    new_params = update(params, max_reward, data)
    return k(new_params)

# Environment handler: cliff-walking.
# Reference: R. Sutton and A. Barto, "Reinforcement Learning: An Introduction", 2020.
def cliff_walking_observe_handler(state: State, action: Action, k, lk):
    reward, new_state = cliff_walking_environment_update(state, action)
    return k(reward, new_state)

def cliff_walking_is_goal(state: State) -> bool:
    """Returns True if state is a goal state."""
    ...

# Usage.
run_qlearning = handle(
    Handler(parameterized=True, random_uniform=random_uniform_handler),
    Handler(parameterized=True, predict=predict_handler, feedback=feedback_handler),
    Handler(observe=cliff_walking_observe_handler),
)(run_episode)

loss_value, (new_rng_key, new_params, new_state) = run_qlearning(rng_key, params)(state, cliff_walking_is_goal)
```

Figure 6: A generic reinforcement learning algorithm in CHOIX (abbreviated for space).

The top-level `run_episode` function is generic and can be instantiated with different environments and agent algorithms through customized handlers. An outer “agent” handler predicts actions and applies feedback, while an inner “environment” handler updates environment state given actions.

Handlers here implement Q-learning (for `predict` and `feedback`) and cliff-walking (for `observe`). The Q-learning `predict_handler` itself uses a `random_uniform` effect for random number generation.

E Deep reinforcement learning

```python
# Reuses functions like `run_episode` from previous example.

@effect
def choose[T](x: T) -> T:
  """Choice effect operation."""

def atari_observe_handler(state: AtariState, action: Action, k, lk):
    reward, new_state = atari_environment_update(state, action)
    return k(reward, new_state)

def ppo_predict_handler(params: Theta, state: AtariState, k, lk):
    # Applies neural network to get per-action logits, then samples one.
    logits, value = forward.apply(params, state)
    ...
    pi = distrax.Categorical(logits)
    action = pi.sample()
    return k(params, action)

def ppo_feedback_handler(params: Theta, data: Feedback, k, lk):
    # Updates agent parameters given feedback.
    new_params = choose(params)
    # PPO loss has three parts: value loss, actor loss, entropy.
    # Intermediate computations are hidden.
    ...
    loss(value_loss)
    ...
    loss(actor_loss)
    ...
    loss(entropy)
    return k(new_params, None)

def choose_optax_handler(opt_state: Opt, params: Theta, k, lk):
    # Gradient-based update via the Optax library.
    grads = jax.grad(lk, argnums=1)(opt_state, params)
    updates, opt_state = optimizer.update(grads, opt_state)
    new_params = optax.apply_updates(params, updates)
    return k(opt_state, new_params)

run_ppo = handle(
    Handler(parameterized=True, random_uniform=random_uniform_handler),
    Handler(parameterized=True, predict=ppo_feedback_handler, feedback=ppo_feedback_handler),
    Handler(parameterized=True, choose=choose_optax_handler),
    Handler(observe=atari_observe_handler),
)(run_episode)

# Usage.
optimizer = optax.adam(learning_rate=...)
opt_state = optimizer.init(...)
loss_value, result = run_ppo(rng_key, params, opt_state)(state, cliff_walking_is_goal)
new_rng_key, new_params, new_opt_state, new_state = result
```

Figure 7: A deep reinforcement learning example in CHOIX.

This reuses the `run_episode` generic function (from Appendix D) with new handlers for `predict` and `feedback`. Here, handlers implement the PPO algorithm [8], abbreviated for simplicity.

In our CHOIX artifact, we use `run_ppo` to train agents on Atari game environments imported from the pgx library[7].

[7]`https://github.com/sotetsuk/pgx`