SOFTWARE 1.0 STRENGTHS FOR INTERPRETABILITY AND DATA EFFICIENCY

Maral JabbariShiviari
CE @ University of the SRBIAU
maraljabbari79@gmail.com

Arshia Soltani Moakhar
CE @ Sharif University of Technology
arshia.soltani2@gmail.com

ABSTRACT

Machine learning has demonstrated remarkable capabilities across various tasks, yet it confronts significant challenges such as limited interpretability, reliance on extensive data, and difficulties in incorporating human intuition. In contrast, traditional software development avoids these pitfalls, offering full interpretability, less data dependency, and easy integration of intuitive decision-making. To have the strengths of both approaches, we introduce the **BasedOn** library. This tool focuses on code written by programmers while providing very simple interfaces to let programmers use machine learning. The **BasedOn** library, leveraging policy gradient methods, offers "learnable" *if statements*.

1 INTRODUCTION

Deep learning has revolutionized various domains, achieving state-of-the-art results in numerous tasks (Yu & Deng (2016), Voulodimos et al. (2018)). However, it inherently grapples with several critical issues. First, despite putting a great extent of effort into explaining these systems' decision-making processes, they are still not well-understood (Doshi-Velez & Kim (2017),Geirhos et al. (2023)). Secondly, they often exploit shortcuts and loopholes in reward functions, resulting in solutions that may be inadequate or biased, as seen in instances of racism and sexism in machine learning models (Zou & Schiebinger (2018)). This limitation is particularly concerning in sensitive fields where fairness and ethical considerations are paramount. Hence, there's a growing interest in integrating human intuition into these systems to guide their decision-making processes more effectively.

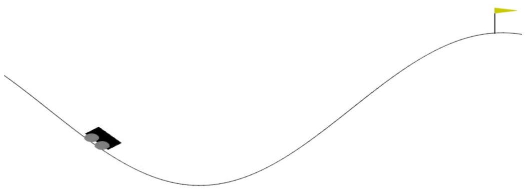
On the contrary, traditional programming is devoid of these issues by design. The explicit nature of code written by programmers ensures full interpretability, predictability, and certifiability of system behavior, making it suitable for sensitive applications where biases like racism must be avoided. Furthermore, traditional software development naturally incorporates the programmer's common sense and intuition, often without the need for extensive data or practice runs. These characteristics suggest that the machine learning ideal of end-to-end training may not be universally applicable, especially in areas requiring high levels of trust and interpretability.

Recognizing the need to blend traditional programming with machine learning, especially in non-differentiable code scenarios (where the function mapping agent parameters to the system loss lack differentiability C.4), we turn to techniques developed within the reinforcement learning (RL) community (Schulman et al. (2017b)). These techniques offer a bridge between traditional software and machine learning. Our approach introduces learnable *If*, *While*, *For*, *Choose from list*, and *Learnable variable* statements into the programmers' toolkit. Programmers can specify the variables involved in decision-making (take *if conditions*), without explicitly defining the condition, thus integrating machine learning insights seamlessly.

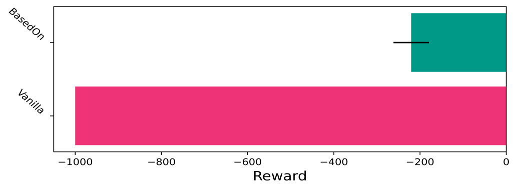
Consider the example of programming a fighting game agent. Normally, programmers define various modes of the agent, such as defense or attack. Here, **BasedOn** can assist in choosing the agent's mode based on variables that programmers think are relevant to the decision like the health and strength of the agent and enemy. This enables programmers to retain their original code while incorporating machine learning techniques in specific aspects of their workflow. The pattern of defining several modes (like Patrolling, Chasing, and Attacking) for automated agents is so common that Unity (Haas (2014)) has developed Finite State Machines (Penny de Byl (2024)) to facilitate such developments. In this example, we explained that the transition condition between these states is usually hand-coded, but BasedOn allows us to employ RL techniques for those conditions.

```
def car(state, f):
    x, v = state
    if x < valley and v > 0: # going down
        return right
    if x > valley and v < 0: # going down
        return left
    if x > valley_threshold:
        if f.if_based_on("R", state):
            return left
        else:
            return right
    if x < valley_threshold:
        if f.if_based_on("L", state):
            return left
        else:
            return right
```

(a)



(b)



(c)

Figure 1: **(a)**: Car decision-making function using **BasedOn**. `Valley` is the x value of valley threshold. **(b)**: MountainCar-V0 visualized state. **(c)**: average reward of the algorithm presented in part (a) compared to vanilla policy gradient, showing the usefulness of programmer intuition. Models were trained on 500 episodes each for 1000 steps. Mean and standard deviation are averaged across 20 runs.

2 BasedOn

The **BasedOn** library presents a straightforward, intuitive interface for integrating Reinforcement Learning into traditional programming paradigms. It consists of a main class named **BasedOn**, and policy networks for each instance of this class. To use **BasedOn** for decision-making, the programmer should use `instance.if_based_on(model_name, inputs)`, where inputs are decision factors and `model_name` allows model to be reused across the code. We include other interfaces of **BasedOn** in the appendix.

Other key functionalities include `train` and `eval` for switching between training and evaluation, `record reward` for providing learning feedback, and `episode ends` for triggering weight updates at the end of each episode. This design combines traditional programming's clarity with the adaptability of machine learning, simplifying the integration of advanced decision-making into software.

3 Toy Experiments

To illustrate the practical application of programmer intuition and to showcase the proposed library, we will utilize the MountainCar-v0 environment from OpenAI Gym. MountainCar-v0 is a simulation where a car must be driven up a steep mountain, with a weak engine to ascend the slope directly. Therefore, the challenge is to build enough momentum by driving back and forth to reach the goal at the top of the mountain. In Fig. 1 (b), you can see a sample state of this task. The car should output *left*, *no acceleration*, or *right* in each step, which shows the direction in which the car accelerates. The car reward is -1 in each step if it does not reach the goal.

A simple intuition is that if the car is going down, it is going down to gain momentum; therefore, we should accelerate its speed in the same direction. On the other hand, if the car is going up, the decision between returning down or continuing up is not easy, and we leave that for reinforcement learning to solve. This intuition could be implemented in the **BasedOn** library as Fig 1 (a). The benefit of using this intuition is also evident according to Fig 1 (c). To give context to the reader, we should note that the State-of-the-art RL algorithm could achieve lower losses. For instance PPO (Schulman et al. (2017a)) achieves -108.2 loss, DQN (Mnih et al. (2013)) achieves -103.4, ARS (Mania et al. (2018)) achieves -123.3 and TRPO (Schulman et al. (2015)) achieves -112.1 loss.

4 Conclusion

In summary, our **BasedOn** library introduces an innovative approach to incorporating Reinforcement Learning (RL) into programming, emphasizing ease of use and intuitive design for programmers. This methodology not only fosters more interpretable systems but also has the potential to lessen data dependencies, albeit with a trade-off in optimality. By blending programmer intuition with machine learning, **BasedOn** sets a new paradigm in software development.

URM Statement

The authors acknowledge that all authors of this work meet the URM criteria of ICLR 2024 Tiny Papers Track.

References

Itai Caspi, Gal Leibovich, Gal Novik, and Shadi Endrawis. Reinforcement learning coach, December 2017. URL https://doi.org/10.5281/zenodo.1134899.

Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G. Bellemare. Dopamine: A Research Framework for Deep Reinforcement Learning. 2018. URL http://arxiv.org/abs/1812.06110.

Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

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.

The garage contributors. Garage: A toolkit for reproducible reinforcement learning research. https://github.com/rlworkgroup/garage, 2019.

Robert Geirhos, Roland S Zimmermann, Blair Bilodeau, Wieland Brendel, and Been Kim. Don't trust your eyes: on the (un) reliability of feature visualizations. *arXiv preprint arXiv:2306.04719*, 2023.

Sergio Guadarrama, Anoop Korattikara, Oscar Ramirez, Pablo Castro, Ethan Holly, Sam Fishman, Ke Wang, Ekaterina Gonina, Neal Wu, Efi Kokiopoulou, Luciano Sbaiz, Jamie Smith, Gábor Bartók, Jesse Berent, Chris Harris, Vincent Vanhoucke, and Eugene Brevdo. TF-Agents: A library for reinforcement learning in tensorflow. https://github.com/tensorflow/agents, 2018. URL https://github.com/tensorflow/agents. [Online; accessed 25-June-2019].

Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

J.K. Haas. A history of the unity game engine. *Digital Media Arts),*, 8(2):104–124, 2014.

Jonathan Ho, Jayesh K. Gupta, and Stefano Ermon. Model-free imitation learning with policy optimization, 2016.

Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*, 2020. URL https://arxiv.org/pdf/1809.02627.pdf.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Jens Kober and Jan Peters. Policy search for motor primitives in robotics. *Advances in neural information processing systems*, 21, 2008.

Sergey Levine. Reinforcement learning and control as probabilistic inference: Tutorial and review. *arXiv preprint arXiv:1805.00909*, 2018.

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.

Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning, 2018.

Eric Mitchell, Charles Lin, Antoine Bosselut, Chelsea Finn, and Christopher D Manning. Fast model editing at scale. *arXiv preprint arXiv:2110.11309*, 2021.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937. PMLR, 2016.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Unity Technologies Penny de Byl. Finite state machines, 2024. URL `https://learn.unity.com/project/finite-state-machines-1`.

Matthias Plappert. keras-rl. `https://github.com/keras-rl/keras-rl`, 2016.

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`.

Reuven Y Rubinstein and Dirk P Kroese. *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation, and machine learning*, volume 133. Springer, 2004.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pp. 1889–1897. PMLR, 2015.

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

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

Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, Eftychios Protopapadakis, et al. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018, 2018.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

Dong Yu and Lin Deng. *Automatic speech recognition*, volume 1. Springer, 2016.

Ningyu Zhang, Yunzhi Yao, Bozhong Tian, Peng Wang, Shumin Deng, Mengru Wang, Zekun Xi, Shengyu Mao, Jintian Zhang, Yuansheng Ni, et al. A comprehensive study of knowledge editing for large language models. *arXiv preprint arXiv:2401.01286*, 2024.

James Zou and Londa Schiebinger. Ai can be sexist and racist—it's time to make it fair, 2018.

A BasedOn limitations

BasedOn tries to utilize programmer intuition and integrate it into the decision-making process. While such intuition makes the system more reliable, interpretable, and data-efficient, it comes with its limitations. It has been shown that end-to-end training results in more optimal solutions in case of accuracy, for instance. In complex scenarios, programmers may not have any intuition about the solution or have wrong intuitions, which could make the proposed library useless in the first case or hurt the performance in the latter.
We should also note that traditional programming is not bias-free. The software design could be the source of gender bias in a program. However, spotting biases is significantly easier given that traditional software is interpretable, and their decision-making is explainable. In addition, traditional software is debuggable, so after spotting a bias in the code, programmers could debug the code and remove the bias. This is in contrast to ML models, where one needs to edit the training data and then train a model from scratch to edit a behavior in the model. Many researchers are working on making models debuggable (Zhang et al. (2024); Mitchell et al. (2021)), but they cannot guarantee the model is edited completely; therefore, their usage in sensitive areas remains limited.

B Related Works

There are several RL libraries aiming to reduce the difficulties of working with RL models like TF-Agents (Guadarrama et al. (2018)), RLLIB (Liang et al. (2018)), Dopamine (Castro et al. (2018)), Keras-RL (Plappert (2016)), Garage (garage contributors (2019)), Unity ML-Agents (Juliani et al. (2020)), Coach (Caspi et al. (2017)), and Stable Baselines3 (Raffin et al. (2021)).

TF-agents (Guadarrama et al. (2018)): In this library, agents need an environment from a specific class for training. If the intent is to use the agent for a learnable if statement within the code, the programmer needs to wrap the rest of the code in this class and implement the required methods for the environment class. This would require changing a significant amount of code. The situation where the programmer would use RL for several decisions in the code, for example for nested learnable ifs, is even more challenging. The programmer will have to implement additional complex logic to ensure agents are pinged correctly. Since in nested if statements, the execution of the inner if statement depends on the results of the outer if statement.
Garage (garage contributors (2019)): Similar to the TF-agents library, this library needs an environment for training the agent. This library defines its-self as a toolkit for developing and evaluating reinforcement learning algorithms and an accompanying library of state-of-the-art implementations. They offer 22 of the most popular RL algorithms, including Behavioural Cloning (Ho et al. (2016)), Cross-Entropy Method (CEM) (Rubinstein & Kroese (2004)), DDPG (Lillicrap et al. (2015)), Episodic Reward Weighted Regression (Kober & Peters (2008)), Soft Actor-Critic (Haarnoja et al. (2018)), RL^2 (Duan et al. (2016)), Twin Delayed Deep Deterministic (Fujimoto et al. (2018)), DQN (Mnih et al. (2013)), REINFORCE (Williams (1992)), and PPO (Levine (2018); Schulman et al. (2017a)).
Stable Baselines3: Raffin et al. (2021): With this library, it is easier to implement agents making decisions and traditional code alongside it. However, since RL agents are the central focus of the library, each agent prediction and output is an environment step that should result in a reward. In our design, multiple RL agents' predictions are gathered, forming the decision together. Therefore, our design has no one-to-one prediction, step, or reward relation. Stable Baselines3 also offers a variety of RL methods like A2C (Mnih et al. (2016)) and TRPO (Schulman et al. (2015)).

The differences and advantages of our library come from the different purposes it is designed for. Most previous libraries were designed to make using different state-of-the-art RL algorithms easier in applications where the RL agent does the decision-making completely. However, in our design, we noticed that it is important that the programmer makes some of the decisions, and some should made by the RL agents.

C Learning algorithm

In this section, we explain the learning algorithm of **BasedOn**. First, we explain a notation and then two possible learning algorithms; one for differentiable environments and one for non-differentiable environments.

C.1 Implementation Detail

For implementing **BasedOn** we used Pytorch Paszke et al. (2019) as a backbone. To make the library easier to use the programmers do not need to specify the model architecture. We use a simple MLP with one hidden layer with 32 neurons as the default model. Neurons use ReLU activation as their activation function. **BasedOn** library will build models and add its parameters to the default optimizer. We used Adam Kingma & Ba (2014) as the default optimizer with a Learning rate of 0.01.

C.2 Notation

Let us assume the learning process is divided into episodes, and each episode consists of multiple steps, resulting in a reward from the environment. In each step, the agent makes some decisions before taking action, which we refer to as $d^e_{s,0}$, $d^e_{s,1}$, $d^e_{s,2}$, $d^e_{s,3}$, etc. $d^e_{s,i}$ refers to the i^{th} decision in step s of episode e. Note that having i in the formulation means in each step, the model could make several decisions. For example, by passing several learnable *if statements*. We consider r^e_s as the reward of step s at episode e.

C.3 Non Differentiable Environment

Non-Differentiable environment is the kind of environment we investigate in the main body. It means we can not calculate the derivative of the agent loss w.r.t the agent parameters even if probabilistic decisions (e.g., learnable ifs, whiles, fors) are fixed. In these cases, we can use policy gradient as the learning process.

In policy gradient, the model assigns a probability to the taken action $\pi(des_{s,i}|state)$. The update rule is

$$\theta_{k+1} = \theta_k + \alpha_k \sum_{s=0} \nabla_\theta \log \pi_\theta(d_s|state) A_s, \tag{1}$$

where

$$A_s = \sum_{s'=s}^{\text{episode length}} r_s * \gamma^{s'-s}. \tag{2}$$

Therefore, in **BasedOn** for each decision, the model outputs probabilities for different potential decision outcomes, and the decision is taken based on those probabilities. However, we might make several decisions in a single step, but we can rewrite the learning rule in Eq. 1 as

$$\theta_{k+1} = \theta_k + \alpha_k \sum_{s=0} \nabla_\theta (\log \prod_i \pi_\theta(d_{s,i}|state)) A_s = \tag{3}$$

$$\theta_k + \alpha_k \sum_{s=0} \nabla_\theta (\sum_{i=0} \log \pi_\theta(d_{s,i}|state)) A_s. \tag{4}$$

To use this formulation for updating weights we store the sum of log probabilities of decisions till a new reward arrives. By the arrival of a new reward, we store the reward and make a new variable for storing the future log probabilities. By the end of the episode, we first calculate A using 2 and then update the weights using 4.

C.4 Differentiable Environment

In some cases, the environment is differentiable which means we can calculate the derivative of the agent loss w.r.t the agent parameters given that probabilistic decisions (e.g., learnable ifs, whiles, fors) are fixed. More specifically, assume $f(x)$ is the deterministic function that maps inputs (x)

```python
def learnable_function(x1, x2, f):
    if f.based_on("d1", x1):
        w3 = f.get_learable_variable("w3")
        return x1 * w3
    else:
        w4 = f.get_learable_variable("w4")
        return x2 * w4
```

(a)

```python
def ground_truth(x1, x2):
    if x1 * math.pi - 1 > 5:
        return x1 * 4.569
    else:
        return x2 * 8.598
```

(b)

Figure 2: (a): the learnable function using **BasedOn**. (b): the function producing ground truth value. The learned function leaned similar parameters to the ground truth. $W3$: 4.5602, $W4$: 8.5824, if_w: 8.0584, if_bias:-15.2819, which results in the exact functionality.

to outputs with all BasedOn decisions being fixed (e.g., Results of all learnable if statements are determined). Then, in a differentiable Environment, we can calculate $\partial_\theta L_\theta(f(x))$. In these cases, we can use a different and more efficient approach. In these cases, the agent makes some decisions during the run which we show by f. Then Assuming the trace of the code is fixed we can calculate the output using the inputs as $f(x)$ where x is the input. In this case, the loss function is given by:

$$\mathcal{L} = \sum_{(x,y)} \sum_{f \in \mathcal{F}} p_\theta(f) L_\theta(f(x)), \tag{5}$$

where

- $p_\theta(f)$ is the probability of making the decision;
- $f(x)$ is the outcome of the input x, with this decisions;
- $L_\theta(f(x))$ evaluates the loss (subject to all parameters θ involved in the computation).

The parameter gradient is given by

$$\begin{aligned}\partial_\theta \mathcal{L} &= \sum_{x} \sum_{f \in \mathcal{F}} \partial_\theta p_\theta(f) L_\theta(f(x)) + p_\theta(f) \partial_\theta L_\theta(f(x)) \\ &= \sum_{x} \sum_{f \in \mathcal{F}} p_\theta(f).\partial_\theta(\ln p_\theta(f)) L_\theta(f(x)) + p_\theta(f) \partial_\theta L_\theta(f(x))) \\ &= \sum_{x} \sum_{f \in \mathcal{F}} p_\theta(f)(\partial_\theta(\ln p_\theta(f)) L_\theta(f(x)) + \partial_\theta L_\theta(f(x)))\end{aligned} \tag{6}$$

Using this formulation we can sample $\partial_\theta \mathcal{L}$ by sampling decisions (f) according to their probability $(p_\theta(f))$. Sampling from $\partial_\theta \mathcal{L}$, we can calculate its mean and update weight using this value.

In figure 2 we present an example of using **BasedOn** in this mode.

D Additional Interfaces

While we only present learnable *if statement*, the framework could offer many other interfaces. For example, *choose (options, inputs)* chooses an option based on the provided inputs in a learnable way.

Or an interface closer to the normal RL could be *learnablefunction(bound, inputs)* where the bearable function learns to choose a real number within the bound based on inputs to maximize the reward.