BF++: A LANGUAGE FOR GENERAL-PURPOSE PROGRAM SYNTHESIS

PREPRINT, COMPILED NOVEMBER 29, 2021

Vadim Liventsev[1*], **Aki Härmä**[2], **and Milan Petković**[3]

[1,3]Eindhoven University of Technology
[1,2,3]Philips Research Eindhoven

ABSTRACT

Most state of the art decision systems based on Reinforcement Learning (RL) are data-driven black-box neural models, where it is often difficult to incorporate expert knowledge into the models or let experts review and validate the learned decision mechanisms. Knowledge-insertion and model review are important requirements in many applications involving human health and safety. One way to bridge the gap between data and knowledge driven systems is program synthesis: replacing a neural network that outputs decisions with a symbolic program generated by a neural network or by means of genetic programming. We propose a new programming language, BF++, designed specifically for automatic programming of agents in a Partially Observable Markov Decision Process (POMDP) setting and apply neural program synthesis to solve standard OpenAI Gym benchmarks. Source code is available at https://github.com/vadim0x60/cibi

***K*eywords** Reinforcement Learning · Program Synthesis · Programming Languages

1 INTRODUCTION

Reinforcement Learning (RL) has been applied successfully in fields like Energy, Finance and Robotics [1]. However, traditional approaches to Reinforcement Learning involve black box models that preclude any exchange of knowledge between experts and ML algorithms. In safety-critical fields[2] like Healthcare [2] the ability to understand the decision algorithms induced by artificial intelligence, as well as to initialize the system using expert knowledge for an acceptable baseline performance, is required for acceptability.

In this work we focus on an alternative approach for RL based on program induction, known as Programmatically Interpretable Reinforcement Learning [3]. We introduce **BF++**, a new programming language tailor-made for this approach (section 4.1). We then demonstrate that neural program synthesis with **BF++** can solve arbitrary reinforcement learning challenges and gives us an avenue for knowledge sharing between domain experts and data-driven models via the mechanism of *expert inspiration* (section 5.5).

2 BACKGROUND

In this paper we define a Reinforcement Learning environment as Partially Observable Markov Decision Process [4, 5]: when at step i the agent takes action $a_i \in A$ it has an impact on the state of the environment $s_i \in S$ via distribution $p_s(s_{i+1}|s_i, a_i)$ of conditional probabilities of possible subsequent states. State is a latent variable that the agent cannot observe. Instead, the agent can see an observation $o_i \in O$ which is a random variable that depends on the latent state via distribution $p_o(o_i|s_i, a_i)$. A, S and O are sets of all possible actions, states and observations respectively. Finally, at every step the agent observes a reward $r_i = R(s_i, a_i)$

[2]Safety requirements in healthcare are the main motivation for our research. However, in this paper we use conventional OpenAI Gym benchmarks to enable comparison between methods

Given this limited toolset, without full (or any) prior knowledge of how the agent's actions influence the the environment (distributions $p_s(s_{i+1}|s_i, a_i)$ and $p_o(o_i|s_i, a_i)$), the agent has to come up with a strategy that will maximize n-step return $R_n = \sum_{t=i}^{n} r_t$ where n is the agent's planning horizon. It is, in the general sense, a hyperparameter, however if an environment has a limit on how many steps an episode can last, it is reasonable to set n equal to the step limit.

Conventional solutions [6] introduce a parametrized *policy function* $\pi_\phi(a|s)$ that defines agent's behavior as a probability distribution over actions and/or function $Q_\phi(a|s)$ that defines what R_n the agent is expecting to receive if they take action a. Parameters ϕ are learned empirically, using gradient descent or evolutionary methods [7, 8].

This approach has been applied extensively and with great success [9] in Partially Observable Markov Decision Process (POMDP) settings, however it does have major limitations:

1. The agent is defined as stateless. As such, when making a decision a_i the agent is unable to take into account any observations it made prior to step i. Long-term dependencies like "this patient should not receive this drug since she has shown signs of allergy when this drug was administered to her 17 iterations ago" cannot be captured by a memoryless model.

2. The agent is represented as a set of model weights ϕ, often with millions of parameters. Such a program can be used as a black box decision system, but domain experts are unable to understand and/or make their contributions to the agent's programming.

In this paper, we address these limitations by representing an RL agent with a program in a specialized language, to be introduced in section 4.1, as opposed to π_ϕ and Q_ϕ

***correspondence:** v.liventsev@tue.nl

3 Related work

Despite Program Synthesis being one of the most challenging tasks in Computer Science, many solutions exist, see [10]. They can be roughly classified by what kind of data they make use of.

One can leverage large datasets of code snippets annotated with natural language like CoNaLa [11] and treat program synthesis as a machine translation task [12]

Given a dataset of program inputs and expected outputs one can search for programs that satisfy given examples [13, 14] using techniques like neural-guided program search [15]. One can also generate input-output pairs artificially [16].

Models like Neural Turing Machines [17], Memory Networks [18] and Neural Random Access Machines [19] are also trained with input-output pairs and even though they don't explicitly generate code, they fit the definition of program.

One can use their own domain knowledge to write a *sketch* of the necessary program in one of specialized programming languages [20] that let the developer leave out certain parameters to be selected via machine learning.

But in this work our goal is to synthesize programs with no training data - only an environment where the program can be tested. This task is typically tackled with neural program synhtesis [3, 21] or genetic programming [22] in a domain-specific language, where the reward function of the POMDP is known as the *fitness function*. However, to the best of our knowledge, there is no programming language for POMDP settings specifically. Because of this, applications of genetic programming for Reinforcement Learning challenges have been limited and the system introduced by Abolafia *et al* [21], for example, only supports non-interactive programs, i.e. a program is a function from an input string to an output string.

4 BF++

4.1 BF syntax

Abolafia *et al* [21] picked BF[3] [23] as their language for program synthesis for the following reasons:

- In industry-grade programming languages like Python or Java program code can contain a very large variety of characters since any of the 143859 Unicode [24] characters can be used in string literals. In BF, however, only 8 characters can be used: they can be one-hot-encoded with vectors of size 8.
- BF's simple syntax means that an arbitrary string of valid characters is likely to be a valid program. In more complex languages, most possible strings result in a syntax error. A generative model being trained to write programs in such a language risks being stuck in a long exploration phase when all the programs it generates are invalid and it has no positive examples in the dataset.
- Despite all of the above, it is a Turing-complete language.

[3]Brainfuck

The simplicity of the language also means that it is relatively easy to develop a compiler that translates programs from an industry-standard programming languages like Java and Python to BF thus making use of the expert knowledge existing in those languages.

In the current paper, we introduce an extended version of the original BF language, BF++. As explained below, the extensions to the original BF syntax are particularly useful in the RL use cases.

BF's runtime model is inspired by the classic Turing Machine [25]: at any point during the program's execution, the state of the program consists of:

- An infinite[4] tape of cells T where each cell holds an integer number.
- A *memory pointer* p_T that points to a certain cell in the tape (*active cell* T^{p_T}).
- A string of characters C that represents program code.
- A *code pointer* p_C pointing to a character about to be executed.

The code pointer starts at the first character, then this character gets executed and the pointer is incremented (moved to the next character). There are 8 possible characters:

>	Move the memory pointer one cell right. $p_T := p_T + 1$
<	Move the memory pointer one cell left. $p_T := p_T - 1$
+	Increment the *active cell*. $T^{p_T} := T^{p_T} + 1$
-	Decrement the *active cell*. $T^{p_T} := T^{p_T} - 1$
.	Write T^{p_T} from the *active cell* to the *output stream*[5]
,	Read x from the *input stream* to the *active cell*. $T^{p_T} := x$
[	If the *active cell* $T^{p_T} = 0$, jump (move p_C) to the matching].
]	If the *active cell* $T^{p_T} \neq 0$, jump (move p_C) to the matching [

[and] commands constitute a loop that will be executed repeatedly until the *active cell* becomes zero. They are also the only way to write a BF program with a syntax error: a valid BF program is one that doesn't contain non-matching [or]

4.2 Negative values

In **BF** memory cells T^i hold non-negative values only. In **BF++** $T^i \in \mathbb{Z}$, a negation operator ~ is introduced and operators [] are redefined to loop while the *active cell* is non-positive, i.e.

~	If the *active cell* $T^{p_T} := -T^{p_T}$.
[	If the *active cell* $T^{p_T} \geq 0$, jump (move p_C) to the matching].
]	If the *active cell* $T^{p_T} < 0$, jump (move p_C) to the matching [

This decision was taken because negative observations are common in control problems (see section 5) as is branching on whether the observed value is positive or negative.

[4]If you happen to be executing a BF program on a computer with finite memory, the tape will be finite due to your hardware limitations.

[5]The definition of input and output streams is purposefully underspecified, it may depend on the particular implementation.

4.3 *Non-blocking action operators*

The main issue of **BF** as a language for Reinforcement Learning is its input-output system. It assumes that the program can freely decide on the relative frequency of inputs to outputs. For example, the following program

```
+[.....,]
```

inputs 5 integers, outputs the 5th character it read, then goes back to the beginning and proceeds indefinitely outputting every 5th character it inputs. Thus it assumes a 5:1 frequency of inputs to outputs. If we simply assume that inputs are observations and outputs are actions, such program will not be able to operate in a POMDP environment where I/O frequency is fixed at 1:1 and the agent that has made an observation has to act before it can make the next observation. In other words, operators `.` and `,` are blocking: `.` stops program execution and waits until new input is received to resume execution, `,` stops program execution and waits until there is an opportunity to act in the environment.

To address this, in **BF++** `.` operator is non-blocking. It outputs the current value of the active cell by placing it at the bottom of the *action queue* S - a sequence of integer numbers that represent actions the program is planning to take in the environment. We also introduce a non-blocking operator `!` that places T^{p_T} on top of the action queue.

$$\begin{array}{ll} . & S := S^\frown(T^{p_T}) \\ ! & S := (T^{p_T})^\frown S \end{array} \tag{1}$$

where $\frown$ denotes concatenation of tuples

The program can thus decide by using `.` or `!` whether the newly added action takes precedence over ones already in the queue. As soon as an opportunity to act arises, the top of the action queue (item S^1 or several items $S^1, S^2, \dots,$ see section 5.2) defines which action the program takes and is then removed from the queue. If S^k does not exist (the queue is empty or shorter than k) default value of $S^k = 0$ is assumed.

`,` operator, on the other hand, is blocking. Thus its function is more important than just reading an observation into memory. Executing `,` is when the program moves to the next step of POMDP.

4.4 *Virtual comma*

The system where the only way to proceed to the following iteration is the `,` operator, naively implemented, means that to be successful in any POMDP environment, a program has to contain an infinite loop with a `,` operator. Any program that has a finite number of `,` steps will terminate prematurely in an environment that supports arbitrarily long number of iterations. Since we originally set out to develop a language where most random programs would be valid, this had to be addressed.

We decided to turn any **BF++** program into an infinite loop with a `,` operator by default:

1. Every **BF++** program starts with a virtual `,` operator at address $p_C = -1$: it is executed before all operators in the code of the program, they are indexed starting from $p_C = 0$
2. When the code pointer p_C reaches the end of the program it loops back to the virtual comma $p_C := -1$

Due to the virtual comma, every program starts executing with the initial observation already stored in memory and available for branching/decision-making.

4.5 *Observation discretization*

Another issue complicating applications of **BF** to Reinforcement Learning is that since its memory tape holds only integer numbers its inputs and outputs have to be integer as well. And this issue cannot be fixed simply by replacing an integer tape with a tape of floating point numbers as **BF**'s only operations for manipulating numbers are `+` and `- -` increment and decrement. Non-integer action and observation spaces are fairly common in reinforcement learning tasks hence **BF++** implements coercion mechanisms for reading and writing continuous vectors into discrete memory.

We assume that the vector observation space O is a hypercube defined as an intersection of n separate scalar observation spaces O^k such that

$$o_1 \in O^k_1, o_2 \in O^k_2, \dots, o_n \in O^k_n \Leftrightarrow (o_1, o_2, \dots, o_n) \in O \tag{2}$$

This assumption theoretically excludes some possible observation spaces, but almost all POMDP tasks discussed in the research literature and all OpenAI Gym tasks conform to this assumption.

To write an observation onto the memory tape we the observation vector of size n is aligned with memory cells $T^{p_T}, T^{p_T+1}, \dots, T^{p_T+n-1}$ and turned into an integer with the use of d discretization bins.

$$T^{p_T+k-1} := \min_{\omega \in 1,\dots d | o^k < \tau^k_\omega} \omega \tag{3}$$

If O^k is an interval $O^k = [o_{low}, o_{high}]$, it is split into discretization bins evenly, as in eq. 4:

$$\tau_\omega = \begin{cases} o_{low} + \frac{o_{high} - o_{low}}{d}\omega, \omega = 1, 2, \dots, d-1 \\ +\infty, \omega = d \end{cases} \tag{4}$$

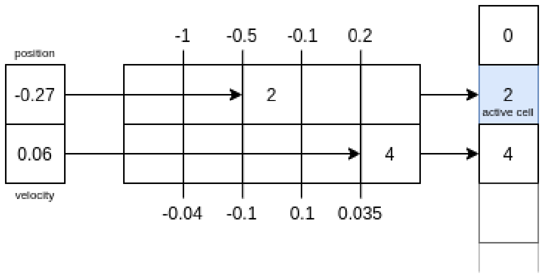


Figure 1: Fluid discretization example in Mountain Car

Some environments, however, have unbounded observation spaces $O^k = (-\infty; +\infty)$, $O^k = (-\infty; o_{high}]$, $O^k = [o_{low}; +\infty)$. This spaces are challenging because the formal description O^k does not in any way reflect the actual underlying distributions of observations. It can be the case, for example, that

$O^k = (-\infty; +\infty)$ but most observations found in the environment fall in the interval $O^k = [42; 43]$. For such observation spaces, **BF++** uses a *fluid discretization* system that learns the true distribution of observations online. The idea was inspired by a work of Touati *et al* [26], although, they assumed that O^k has a finite diameter and didn't support unbounded observation spaces. Initial thresholds τ_ω can be arbitrary. With each new observation, thresholds τ_ω are readjusted so that among h prior observations, roughly ω out of d observations are lower values that τ_ω:

$$\underset{\tau}{\text{minimize}} \sum_{\omega \in 0,1,\ldots d} \left|\frac{\omega}{d} - \frac{\sum_{i' \in i-h, i-h+1, \ldots, i-1} \mathbb{I}(o^k_{i'} < \tau_\omega)}{h}\right| \quad (5)$$

To solve this optimization problem, one has to sort previous d observations in ascending order so that

$$\text{sort} : \{o_i | i \in i-h, i-h+1, \ldots, i-1\} \longrightarrow \{s_i | i \in 1, 2, \ldots, h\} \quad (6)$$

is such a bijection that $s_1 < s_2 < \cdots < s_h$ holds and set

$$\tau_\omega = s_{\lceil \frac{\omega}{d} h \rceil} \quad (7)$$

See figure 1 for a visual example.

This system has 2 hyperparameters: d and h. With a low d a lot of the information observed form the environment is lost, while when d is in the hunderds the generated programs can become very complex. h switches between relative and absolute observations. With a very high h, $\omega = 0$ means that this observation is one of the lowest that can be observed in this environment, with $h = 1$ it means that the observation is lower than the previous one.

High values of h present an additional challenge: how to correctly discretize observation in the first h iterations? We implemented *burn-in*: before training or evaluation we run h iterations of a random agent (see section 4.8) to collect a history of h observations and pick correct thresholds.

4.6 Action coercion

A symmetrical problem arises with actions taken by the agent. Memory tape holds integer numbers $T^k \in \mathbb{Z}$ and any value can be pushed onto the action stack. However, the action that's output to the environment has to belong to a N-dimensional action space A, an intersection of unidimensional action spaces A^k. The "act" operation thus includes a coercion system and is defined as:

$$a^k := \begin{cases} \frac{S^k}{d-1}, A^k = (-\infty; +\infty) \\ a_{\min} + |\frac{S^k}{d-1} - a_{\min}|, A^k = [a_{\min}; +\infty) \\ a_{\max} - |a_{\max} - \frac{S^k}{d-1}|, A^k = (-\infty; a_{\max}] \\ a_{\min} + \frac{(S^k \bmod d)}{d-1} * (a_{\max} - a_{\min}), A^k = [a_{\min}; a_{\max}] \\ S^k, A^k \subset \mathbb{Z} \end{cases}$$
$$S := (S^{N+1}, S^{N+2}, \ldots) \quad (8)$$

4.7 Goto

It is notoriously hard to introduce any kind of branching behavior in **BF** [27]. To facilitate if-then style programs we introduce a *goto* operator `^` defined as

$$p_T := T^{p_T} \quad (9)$$

Note that it is not a `goto;` in the traditional C sense, since the memory pointer is being moved, not the code pointer. Still, it lets the agent preemptively store potential actions in memory cells and than branch between this actions based on the observation.

4.8 Random number generator

Operator `@` writes a random number into the *active cell*. A random agent is often used as a starting point for exploration and in **BF++** a random agent can be implemented as `@!`

4.9 Shorthands

With all the commands we introduced in sections 4.1 - 4.7 it is still surprisingly hard to encode relatively simple decisions like "add action 5 to the top of the action queue":

```
[>]+++++!
```

This program moves the memory pointer right until it hits a cell that contains zero, increments it five times, and then pushes T^{p_T} to the top of the action queue. It also loses the current value of the memory pointer which might be meaningful. Our experiments have shown that it takes a very long time for the neural model to learn to write this kind of combinations.

To mitigate this issue we introduce *shorthands*: commands `01234` mean "write the respective number (0,1,2,3 or 4)" into the *cell* and commands `abcde` mean "move the memory pointer to cell a,b,c,d or e" where cells a,b,c,d and e are the first 5 cells in the memory tape. We intentionally made the number of *shorthands* equal to discretization constant $d = 5$. Due to our method of discretization of continious action spaces (see sections 4.5, 4.6) the program will often encounter situations when it can choose between d different actions and thanks to shorthands taking them can be encoded as `1!`, `2!`, ...

4.10 Summary

In total (assuming 5 shorthands) **BF++** has 22 commands:

```
><^@+~-[].,!01234abcde
```

Commands `@^~01234abcde` are considered optional and can be disabled if the task at hand calls for it. The number of shorthand commands can be increased or decreased.

Observation discretization and action coercion techniques built into the language mean that **BF++** is compatible with any POMDP environment. However, in practice, there is one important limitation: the complexity of the program required to operate in an environment is directly proportional to dimensionality of it's action and observation spaces A and O. If, for example the observation space is 10000-dimensional, once an observation is read onto tape T it takes 9999 > operators to reach second to last observation. Thus, in practice, **BF++** should be used with low-dimensional POMDPs.

An extension of our methodology to high-dimensional POMDPs (such as Atari games [28], where the observation is a matrix of pixels on simulated game screen) can be achieved by adding a scene encoder neural network that maps the observed image to a low-dimensional vector as proposed in [29].

5 Experimental setup

5.1 Hypotheses and goals

Our experiments were designed to test the following hypotheses:

H_1 **BF++** can be used in conjunction with a program synthesis algorithm to solve arbitrary reinforcement learning challenges (POMDPs)

H_2 **BF++** can be used to take a program written by an expert and use program synthesis to automatically improve it

H_3 **BF++** can be used to generate an interpretable solutions to Reinforcement Learning Challenges that experts can learn from

H_4 Optional commands `@^~01234abcde` introduced for convenience make it easier for experts to write programs in **BF++**

H_5 Optional commands `@^~01234abcde` improve the quality of programs synthesised by neural models

Hence we

1. Pick several commonly studied reinforcement learning environments
2. Employ an expert[6] to write **BF++** programs to solve them
3. Develop a program synthesis model following from [21]
4. Compare the best programs generated by the model with expert programs in terms of program quality
5. Perform ablation studies: remove some of the optional commands from the language (resulting language is called **BF+**), remove the expert program from the model's program pool, compare program quality
6. Perform case studies: analyze programs generated by the model to gain insight into how the model approached the problem

5.2 Environments

We evaluate our framework on 4 low-dimensional (see section 4.10) POMDPs sampled from OpenAI Gym [30] leaderboard[7]:

1. **CartPole-v1** [31]. A pole is attached to a cart. which moves along a frictionless track. The agent observes cart position, cart velocity, pole angle and pole velocity at tip. The goal is to keep the pole upright by applying force between -1 and 1 to the cart. At every step the agent receives a +1 reward for survival. The episode terminates when the pole inclines too far.
2. **MountainCarContinuous-v0** [32]. A car is on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain consuming a minimal amount of fuel by controlling the engine, setting it's torque in the range $[-1; 1]$; however, the engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. We picked MountainCarContinuous-v0 as opposed to MountainCar-v0 to demonstrate the performance of our discretization system.
3. **Taxi-v3** [33]. There are 4 locations (labeled by different letters) and the goal is to pick up the passenger at one location and drop him off in another in as few timesteps as possible spending as little fuel as possible.
4. **BipedalWalker-v2**. A simulated 2D robot with legs has to learn how to walk. Moving rightwards is rewarded, falling is penalized. Observation vector consists of speeds, angular speeds and joint positions collected by the robot's sensors. These observations do not, however, include any global coordinates - they can only be inferred from sensor inputs. With action vector of size 4 the agent controls speeds of the robots hip and knee motors.

[6]first author of this paper

[7]https://github.com/openai/gym/wiki/Leaderboard

5.3 Hyperparameters

For observation discretization (section 4.5) we picked $d = 5$ (so that it's equal to the number of shorthands) and $h = 500$ for our experiments, hence when the observation is among the highest 20% of the last 500 observations it is written into memory as 4 while if it falls between 40-th and 60-th percentiles it is 2.

5.4 Expert programs

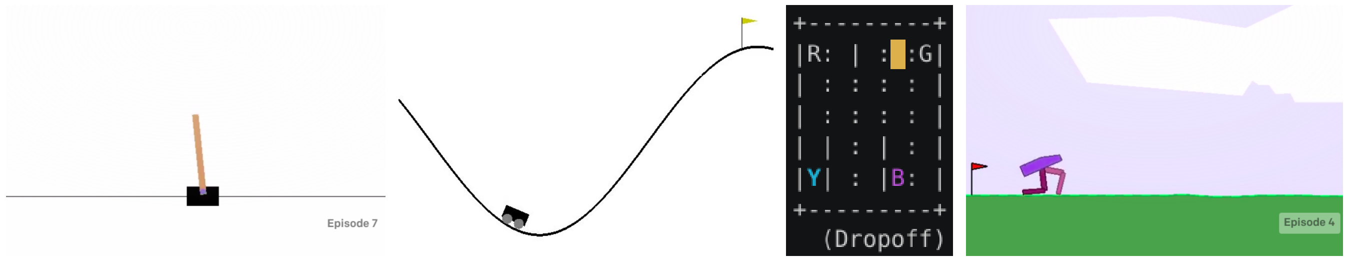
For **CartPole** we wrote 2 programs. One completely ignores all observations and just alternates between "move right" and "move left":

```
0!,1!
```

Another calculates the difference between velocity of the cart and angular velocity of the pole. If it's positive, the cart is pushed to the right (the cart has to catch up with the pole), if it's negative the cart is pushed to the left, if zero it is pushed randomly:

```
[a0>0>0>0>0>@>1>1>1>1>1>,>[->>-<<]>>+++++^!1]
```

The first part of this program sets up an action map on the tape where every possible value of the velocity differential has a respective cell with 0, 1 or (in the center) random number. Then `[->>-<<]` block does subtraction, +++++ adds 5 to the result, so that it belongs to in 0..10 and not $-5..5$, `^` moves the memory pointer to the correct cell in the action map and `!` puts the action onto the action stack.



(a) CartPole-v1 (b) MountainCarContinuous-v0 (c) Taxi-v3 (d) BipedalWalker-v2

Figure 2: Selected environments, visualized

For **Mountain Car** we wrote an elegant algorithm that reads the observation vector into the tape, goes to the second observation (car velocity) and outputs it as action:

```
>!a
```

In other words, we apply motor torque in the same direction where we're currently headed, thus always accelerating our car. If we're headed right, that helps us get to the destination and if we're headed left that helps us get as high as possible onto the hill so that when direction reverses, the car has more energy to push through the right hill.

For **Taxi** we introduce 2 programs. The first program:

1. Finds the coordinates of the current destination (passenger to pick up or current passenger's destination)
2. Subtracts the current destination
3. Moves in the resulting direction

The problem with this approach is that it always gets stuck when it hits a wall. To compensate for that, the second program alternates between the strategy above (for 5 iterations) and random movements (for 5 iterations) so that it eventually gets unstuck. See source code repository for the programs.

Optional commands `@^~01234abcde` have all been invaluable in developing these programs - a fact in support of H_4. A more rigorous way to confirm it would be employing several human experts to develop programs with and without optional operators, but finding volunteer **BF++** developers has proven difficult.

Developing programs for **Bipedal Walker** is, unfortunately, above our expert's paygrade.

5.5 *Program synthesis model*

In order to train a generative model g to write **BF++** programs we treat the writing process as a reinforcement learning episode in its own right [21] . Every character of a program is an action taken by the *writer agent*, the programs are terminated by a NULL character. When the NULL character is written, a *BF++ agent* is created in the target POMDP environment (e.g. CartPole) and sum total of rewards Q collected in that episode is assigned as a reward to the *writer agent* for the NULL character. All other characters are rewarded with zero.

The *writer agent's* policy is modeled with an LSTM [34] neural network and is trained with a modified version of REINFORCE

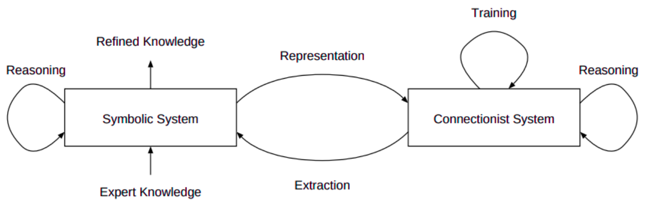


Figure 3: Neural-symbolic learning cycle [36]

[35] algorithm. While standard REINFORCE optimizes Policy Gradient:

$$O_{\text{PG}}(\phi) = \mathbb{E}_{\pi(\mathbb{C};\phi)}(Q) \tag{10}$$

where ϕ are LSTM parameters, C - program, Q - reward obtained by the program in target environment,

we optimize

$$O(\phi) = O_{PG}(\phi) + O_{PQT}(\phi) \tag{11}$$

where

$$O_{\text{PQT}} = \frac{1}{K}\sum_{k=0}^{K} \log \pi(C_k;\phi) \tag{12}$$

where C_1 is the best (highest Q) known program, C_2 - second best, ...

Intuitively, both $O_{\text{PG}}(\phi)$ and $O_{\text{PQT}}(\phi)$ when optimized update the weights of the LSTM so that programs that we have found to be successful are more likely. But Policy Gradient weighs programs proportionately to their respective rewards while PQT creates a *priority queue* of the *best known programs* and assigns a high importance to them and zero to the rest.

O_{PQT} component has been shown to have "a stabilizing affect and helps reduce catastrophic forgetting in the policy" [21]. In addition to this, we use O_{PQT} to implement **expert inspiration**. By default, the *priority queue* of the *best known programs* is initialized as an empty set. But if expert-written programs are available, it can be prepopulated with these programs that act as useful positive examples for teaching the *writer agent*. This approach is used to incorporate programs from section 5.4 and transfer knowledge from experts to the neural developer.

This approach to **expert inspiration** follows what's known as neural-symbolic learning cycle, displayed in figure 3 - expert knowledge is represented symbolically, in terms of a **BF++** program, then a neural network is trained to generate this program, effectively translating the expert knowledge from symbolic into connectionist format (*representation*), the neural network learns from reinforcement how to solve the task better than the expert (*training*). Unlike in most neural-symbolic systems [37] that extract knowledge from connectionist systems with algortihms like TREPAN [38] or JRip extraction [39], the *extraction* step is trivial since the neural network outputs a symbolic program directly.

In all experiments below, the *writer agent*'s LSTM has hidden size of 50, batch size of 4 and is trained with RMSProp [40] optimizer.

5.6 Stopping and Scoring

All experiments were run with an upper limit of 100000 training episodes. Environments other than **Taxi** also used Exponential Variance Elimination [41] early stopping technique - training was stopped when the postive trend in the quality of the best found program stopped, i.e. when the exponential moving average of program quality is lower that it was 1000 episodes ago. Agents for **Taxi** are trained for a fixed number of episodes, because we noticed that in this environment the longest part of the training process is learning to pick up your first passenger and until that happens $Q = -200$ holds.

Once the training process is finished, we take the best known programs and since each of them was only tested once (leading to high variance) we test them again, averaging total rewards over 100 episodes. We use this averaged reward to pick the best program.

5.7 Implementation

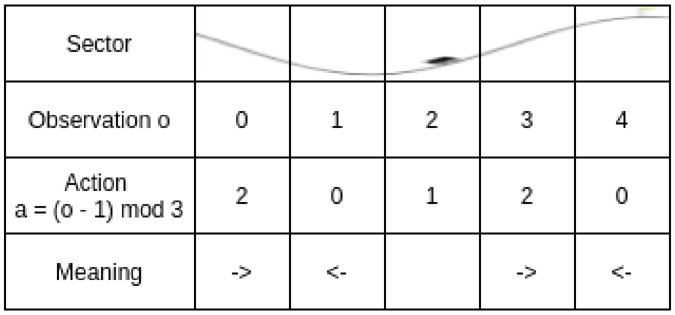
BF++ interpreter and the training system were written in Python with TensorFlow for neural models. GPU resources weren't used, because the performance bottleneck of the system is not backpropagation but rather testing a **BF++** program in the environment, single experiment runtime was between 1 hour (CartPole) and 10 (Taxi).

6 RESULTS

6.1 Quantitative results

Table 1 presents the quality metric (average 100-episode reward) of the best program in every category, compared to that of a fully random agent and the result required to join the OpenAI gym leaderboard for context. Note that the expert programs used a lot of optional operators (shorthands and `@^!`), so it wasn't possible to implement expert inspiration with limited command sets.

These results support (see section 5.1) hypothesis H_1 - we have obtained functional programs for all environments, H_2 - when expert inspiration was used the resulting programs were better than expert programs and better than programs generated without expert inspiration and H_4 - ablation studies for optional operators do indeed show that those operators are useful.



Sector					
Observation o	0	1	2	3	4
Action a = (o - 1) mod 3	2	0	1	2	0
Meaning	->	<-		->	<-

Figure 4: Visual summary of the strategy enacted by `-..` on **Mountain Car**

6.2 Case studies

We have established that the program synthesis model is able to learn from human experts. But can experts learn from the model? (H_3) To confirm this, we offer a detailed explanation of the most successful program of all experiments listed in section 5.

This program scored *91.39* on **Mountain Car**:

```
-..~+
```

The trailing ~ and + do not affect the behavior of the agent: they modify the value of the active cell only for it to be immediately rewritten by the virtual comma (section 4.4) before it has any chance to influence actions. One can think about these commands as inactive genes in the DNA - we have found many resulting programs to contain such commands. If necessary this effect can be accounted for by incorporating program length into the loss function. So this program is equivalent to:

```
-..
```

When the virtual comma is executed, car position and car velocity are read into memory, discretized into integers $0 \dots 4$. The position is read into the active memory cell p_T, while the velocity is in cell $p_T + 1$. Then the active cell is decremented and the resulting number is put onto the action stack twice. There is 1 read operation and 2 write operations to the end of the action stack, which introduces a delay before the actions get executed. When it's time to act, the number on the action stack is coerced to one of the actions possible in this environment (0 for going left, 1 for doing nothing, 2 for right).

A strategy emerges, illustrated on figure 4, in which the car puts "going right" onto the agenda if it's on the far left or the center right of the landscape, puts "going left" onto the agenda when it's on the far right or center left and schedules doing nothing if it's in the center. This strategy helps the car successfully reach the right fringe every time it is applied.

7 CONCLUSIONS

In this paper, we have introduced a new programming language tailored to the task of programmatically interpretable reinforcement learning. We have shown experimentally that this language can facilitate program synthesis as well as knowledge transfer between expert-based systems and data-driven systems.

Table 1: Total episode reward Q achieved by best programs found, averaged over 100 episodes

Environment	CartPole-v1	MountainCarContinuous-v0	Taxi-v3	BipedalWalker-v2
Random agent	9.3	0	-200	-91.92
BF++ expert program 1	20.48	-6.55	-179.49	-
BF++ expert program 2	18.23	-	-150.44	-
BF+ (without shorthands) LSTM	44.55	91.57	-57.93	-91.9
BF+ (without `@^~`) LSTM	48.14	81.16	-42.21	-31.79
BF++ LSTM	71.38	88.41	-199.82	-26.97
BF++ LSTM with expert inspiration	96.64	91.39	-60.65	-
Leaderboard threshold	195	90	0	300

The results in the OpenAI gym test examples show that the proposed system is able to find a functional solution to the problem. In some cases the performance is similar to the best deep learning solution but the obtained program remains still explainable. This is a very encouraging result and suggest that the use of program induction methods may indeed be a viable way towards explainable solutions in RL applications.

We propose the following directions for future work:

1. Develop translation mechanisms between **BF++** and other languages. Potentially, **BF++** can be used as *bytecode* [42] for reinforcement learning. The expert would write a program in a higher-level language and transpile it into **BF++** so that the program then can be improved with reinforcement learning.
2. Use other neural network architectures as well as non-neural evolution methods like genetic programming [43] in conjunction with **BF++**
3. Apply the framework to problems in Healthcare where expert inspiration is important for crossing the AI chasm [44].
4. Use Natural Language Generation techniques to translate the BF++ code automatically to a friendly human-readable text description as in [45, 46].

ACKNOWLEDGEMENTS

This work was funded by the European Union's Horizon 2020 research and innovation programme under grant agreement n° 812882. This work is part of "Personal Health Interfaces Leveraging HUman-MAchine Natural interactionS" (PhilHumans) project: https://www.philhumans.eu

REFERENCES

[1] Yuxi Li. Reinforcement learning applications. *CoRR*, abs/1908.06973, 2019. URL http://arxiv.org/abs/1908.06973.

[2] Chao Yu, Jiming Liu, and Shamim Nemati. Reinforcement learning in healthcare: a survey. *arXiv preprint arXiv:1908.08796*, 2019.

[3] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5045–5054, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL http://proceedings.mlr.press/v80/verma18a.html.

[4] K J Åström. Optimal control of Markov processes with incomplete state information. *Journal of Mathematical Analysis and Applications*, 10(1):174–205, 1965. ISSN 0022-247X. doi: https://doi.org/10.1016/0022-247X(65)90154-X. URL http://www.sciencedirect.com/science/article/pii/0022247X6590154X.

[5] Jr Kramer, J David R. Partially Observable Markov Processes., 1964.

[6] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction, Second edition in progress*, volume 3. 2017. doi: 10.1016/S1364-6613(99)01331-5.

[7] Seyed Sajad Mousavi, Michael Schukat, and Enda Howley. Deep Reinforcement Learning: An Overview. In *Lecture Notes in Networks and Systems*, volume 16, pages 426–440. 2018. doi: 10.1007/978-3-319-56991-8_32. URL https://arxiv.org/abs/.

[8] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017. doi: 10.1109/MSP.2017.2743240.

[9] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[10] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017. ISSN 23251131. doi: 10.1561/2500000010. URL www.nowpublishers.com;.

[11] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pages 476–486. ACM, 2018. doi: https://doi.org/10.1145/3196398.3196408.

[12] Xiaojun Xu, Chang Liu, and Dawn Song. Sqlnet: Generating structured queries from natural language without reinforcement learning. *CoRR*, abs/1711.04436, 2017. URL http://arxiv.org/abs/1711.04436.

[13] Neel Kant. Recent Advances in Neural Program Synthesis. 2018. URL http://arxiv.org/abs/1802.02353.

[14] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 107–126, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336895. doi: 10.1145/2814270.2814310. URL https://doi.org/10.1145/2814270.2814310.

[15] Ashwin K Vijayakumar, Dhruv Batra, Abhishek Mohta, Prateek Jain, Oleksandr Polozov, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. In *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2018. URL https://microsoft.github.io/prose/impact/.

[16] Richard Shin, Neel Kant, Kavi Gupta, Christopher Bender, Brandon Trabucco, Rishabh Singh, and Dawn Song. Synthetic datasets for neural program synthesis. Technical report, 2019.

[17] Wojciech Zaremba and Ilya Sutskever. Reinforcement learning neural turing machines. *CoRR*, abs/1505.00521, 2015. URL http://arxiv.org/abs/1505.00521.

[18] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. In *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, oct 2015. URL http://arxiv.org/abs/1410.3916.

[19] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. In *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 2016.

[20] Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *CoRR*, abs/1608.04428, 2016. URL http://arxiv.org/abs/1608.04428.

[21] Daniel A. Abolafia, Mohammad Norouzi, Jonathan Shen, Rui Zhao, and Quoc V. Le. Neural Program Synthesis with Priority Queue Training. 2018. URL http://arxiv.org/abs/1801.03526.

[22] Milad Taleby Ahvanooey, Qianmu Li, Ming Wu, and Shuo Wang. A survey of genetic programming and its applications. *KSII Transactions on Internet and Information Systems (TIIS)*, 13(4):1765–1794, 2019.

[23] U. Muller. Brainfuck – an eight-instruction turing-complete programming language. Available at the internet address http://en.wikipedia.org/wiki/Brainfuck, 1993. URL http://en.wikipedia.org/wiki/Brainfuck.

[24] Julie D Allen, Deborah Anderson, Joe Becker, Richard Cook, Mark Davis, Peter Edberg, Michael Everson, Asmus Freytag, Laurentiu Iancu, Richard Ishida, et al. The unicode standard. *Mountain view, CA*, 2012.

[25] A M Turing. On computable numbers, with an application to the entscheidungsproblem. a correction. *Proceedings of the London Mathematical Society*, s2-43(1):544–546, 1938. ISSN 1460244X. doi: 10.1112/plms/s2-43.6.544.

[26] Ahmed Touati, Adrien Ali Taiga, and Marc G Bellemare. Zooming for efficient model-free reinforcement learning in metric spaces. *arXiv preprint arXiv:2003.04069*, 2020.

[27] Mats Linander. control flow in brainfuck | matslina, 2016. URL http://calmerthanyouare.org/2016/01/14/control-flow-in-brainfuck.html.

[28] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.

[29] Daiki Kimura. Daqn: Deep auto-encoder and q-network. *arXiv preprint arXiv:1806.00630*, 2018.

[30] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016. URL http://arxiv.org/abs/1606.01540.

[31] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, Sep. 1983. ISSN 2168-2909. doi: 10.1109/TSMC.1983.6313077.

[32] Andrew William Moore. Efficient memory-based learning for robot control. Technical report, 1990.

[33] Thomas G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[34] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[35] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[36] Sebastian Bader and Pascal Hitzler. Dimensions of neural-symbolic integration-a structured survey. *arXiv preprint cs/0511042*, 2005.

[37] Tarek R Besold, Artur d'Avila Garcez, Sebastian Bader, Howard Bowman, Pedro Domingos, Pascal Hitzler, Kai-Uwe Kühnberger, Luis C Lamb, Daniel Lowd, Priscila Machado Vieira Lima, et al. Neural-symbolic learning and reasoning: A survey and interpretation. *arXiv preprint arXiv:1711.03902*, 2017.

[38] Manoel Vitor Macedo França, Artur S d'Avila Garcez, and Gerson Zaverucha. Relational knowledge extraction from neural networks. In *CoCo@ NIPS*, 2015.

[39] Martin Svatoš, Gustav Šourek, and Filip Železný. Revisiting neural-symbolic learning cycle. In *14TH INTERNATIONAL WORKSHOP ON NEURAL-SYMBOLIC LEARNING AND REASONING*, 2019. URL https://sites.google.com/view/nesy2019/home.

[40] T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[41] Vadim Liventsev. vadim0x60/evestop: Early stopping with exponential variance elmination. https://github.com/vadim0x60/evestop, 2021. (Accessed on 01/20/2021).

[42] Wikipedia contributors. Bytecode — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/w/index.php?title=Bytecode&oldid=995026385. [Online; accessed 21-January-2021].

[43] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to genetic programming*. Lulu. com, 2008.

[44] P. A. Keane and E. J. Topol. With an eye to AI and autonomous diagnosis. *NPJ Digit Med*, 1:40, 2018. [PubMed Central:PMC6550235] [DOI:10.1038/s41746-018-0048-y] [PubMed:29618526].

[45] Kyle Richardson, Sina Zarrieß, and Jonas Kuhn. The code2text challenge: Text generation in source code libraries. *CoRR*, abs/1708.00098, 2017. URL http://arxiv.org/abs/1708.00098.

[46] A. LeClair, S. Jiang, and C. McMillan. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 795–806, 2019. doi: 10.1109/ICSE.2019.00087.