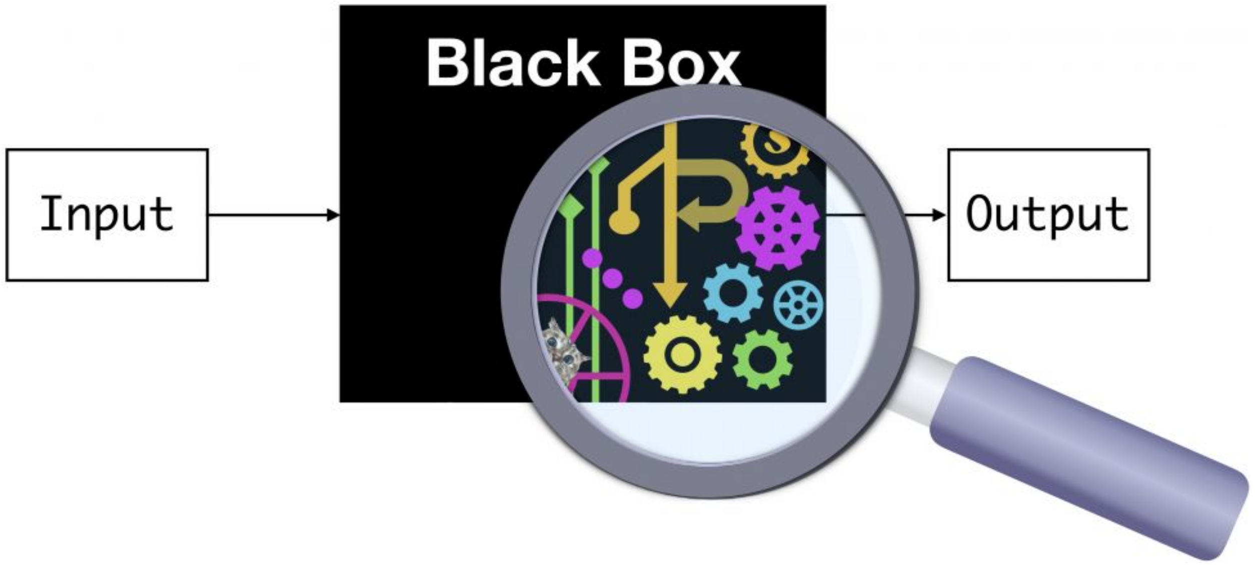
Independent Study

Report-I, 15th January

Submitted by: Vaibhav Garg(20171005), Kartik Gupta(20171018)

AIM



The primary goal of the project is to introduce more interpretability and transparency in RL models which are now becoming increasingly *black box* based due to the advent of Deep RL without making a significant compromise in the performance gained by the SOTA DRL models. The idea is to represent a RL policy using a program written in a DSL which by its very structure/definition will be interpretable and verifiable using symbolic methods. The real challenge is how exactly to find a program which simulates a high performance policy since past attempts show that the space of programs is really large, making the search intractable and also the reward function doesn't exactly lie on a smooth trajectory making the search even more difficult. The first problem is somewhat handled by introducing the notion of a *Sketch,* which helps in limiting the search space by a considerable factor and the latter problem is countered by shifting the search to a *Imitation Learning* based space where the aim is to find a program that resembles behaviour with a high performing DRL model(*a neural oracle*).

Program Synthesis

This project aims to find a program for an already well performing policy black box. Since this means an inevitable search over the program space, program space is constrained using 2 methods.

DSL

First, the authors provide a functional DSL, with very few primitives. This makes it easier to represent policies compactly and canonically, hence making program synthesis easier.

The DSL consists of 2 data primitives

1. Atoms: Any observation, action or number generated during computation
2. Sequences of Atoms: These allow the programs to work on histories which are represented as series of alternate observations and actions

Operators operate on these data primitives. There are 2 classes of operators:

1. Acting on Atoms: standard arithmetic operators like +. -. *, /, act on atoms.
2. Acting on Atom Sequences:
    a. Peek(x, i) : return the ith element of history x
    b. Fold(f, [e_n, e_{n-1}, … e_1]) := f(e_n, f(e_{n-1}, .. f(e_2, e_1)))
        ie, a combinator operating on an input sequence

For control flow, the language supports a simple i statement. We can note here that the if statement itself can be made from the previously defined constructs, and it's explicit inclusion is only a matter of convenience.

The DSL also provides for variables and constants (numbers) with the convention that unbound variables are inputs to the program.

Although strict type checking is not required since all the data primitives are composed of numbers, some amount of type checking is employed to ensure index bounds in sequences, and atoms are not indexed.

Grammar "Sketch"

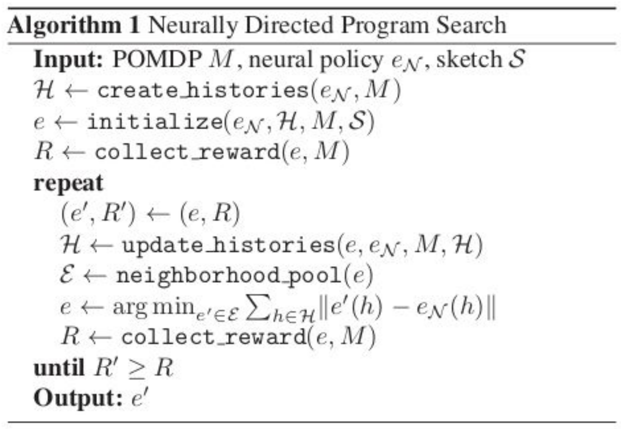
Even though the DSL is small, it can still generate an infinite number of programs, especially given that it allows for arbitrary numeric constants. To further restrict the search space, the authors provide an idea of a grammar "sketch".

A sketch is a Context Free Grammar (CFG) composed using primitives of the DSL. The search is performed only over the constrained space of valid languages generated by the CFG.

Herein lies one issue with this approach. The grammar is composed using the domain expertise of humans, but it might not be the most optimal grammar for the problem at hand. From this point on the algorithm aims to find the most optimal program which can be generated by this grammar, and not the most optimal program that can be written in the DSL to solve the RL problem.

NDPS

The NDPS algorithm is at the heart of this project as it is the one doing the search in the DSL program space, finally outputting the one closest to the *neural oracle.* Below is the pseudo code for the same:



```
Algorithm 1 Neurally Directed Program Search
  Input: POMDP $M$, neural policy $e_{\mathcal{N}}$, sketch $\mathcal{S}$
  $\mathcal{H} \leftarrow$ create_histories($e_{\mathcal{N}}, M$)
  $e \leftarrow$ initialize($e_{\mathcal{N}}, \mathcal{H}, M, \mathcal{S}$)
  $R \leftarrow$ collect_reward($e, M$)
  repeat
    $(e', R') \leftarrow (e, R)$
    $\mathcal{H} \leftarrow$ update_histories($e, e_{\mathcal{N}}, M, \mathcal{H}$)
    $\mathcal{E} \leftarrow$ neighborhood_pool($e$)
    $e \leftarrow \arg\min_{e' \in \mathcal{E}} \sum_{h \in \mathcal{H}} \| e'(h) - e_{\mathcal{N}}(h) \|$
    $R \leftarrow$ collect_reward($e, M$)
  until $R' \geq R$
  Output: $e'$
```

The algorithm takes as input the neural black-box policy, the Sketch and the POMDP itself and outputs a program which best imitates the Neural policy. The first step of the algorithm is to generate some histories that'll serve as interesting inputs for the algorithm, this is a very important step as we'd want these histories to be very diverse making sure that our program performs similar to the oracle in all circumstances, this is similar to making sure that the training data distribution of any deep learning model resembles that of the testing data for proper generalization.

The next step is to generate some initial program that can be improved iteratively, as you can imagine this is a very critical step, a bad initialization could really mess up future performance. Therefore, instead of generating one single initial program a set of good initializations is considered out of which the closest to oracle is finally selected using *some* distance metric.

Finally, the program is iteratively improved in a greedy fashion by choosing from a pool of neighbouring programs the one with minimum distance from the neural oracle. It is important to notice here that the **search space here is not directly related to the reward function and rather optimizing on the distance between the oracle and the DSL program which provides a way for optimization on a smooth surface.** The neighbourhood pool generation in itself isn't a very simple task and needs to be handled carefully making sure all possible branching, template and parameter changes are explored. The algorithm converges when no neighbours give a smaller distance than the current program with the neural oracle.

There is also one more small step called *History Augmentation* which basically updates the interesting inputs to make sure only valid histories are being considered.

Related Work

There is considerable work in the field of program synthesis. The approach above is an adaptation of the SYGUS algorithm [1]

This approach also takes inspiration from classical imitation learning with the crucial difference that here the aim is to obtain the same goal as the oracle, whereas in imitation learning the aim is to mimic every step precisely. Authors also take inspiration about input augmentation from DAGGER algorithm.[2]

An alternative to this approach is to learn a decision tree to mimic the oracle as has been done in VIPER. [3] We believe that the current approach, of generating a program, provides more flexibility as compared to a decision tree, although a direct comparison to verify this claim might be an interesting problem.

Possible issues with the approach

The most likely issue with this approach could be a significant drop in results/accuracy in the DSL program policy from the Deep RL model. This is primarily due to the much larger policy space that is spanned by the Deep RL models compared to the DSL program space which is further constrained by the *Sketch.*

Code availability and extensibility

Code is not provided by the authors and no alternate implementations were found in our research online. As part of our project we will be aiming to provide a library to support research in this direction.

[1] "SyGuS." https://sygus.org/. Accessed 15 Jan. 2021.
[2] "A Reduction of Imitation Learning and Structured Prediction to ...." https://www.ri.cmu.edu/pub_files/2011/4/Ross-AISTATS11-NoRegret.pdf. Accessed 15 Jan. 2021.
[3] "Verifiable Reinforcement Learning via Policy Extraction." https://arxiv.org/abs/1805.08328. Accessed 15 Jan. 2021.