Imitation Learning:
Explainable AI Assignment 2

Dawn Estes McKnight

March 20, 2020

1 Introduction

Computing has seen the rise of many sophisticated programs for playing games that humans enjoy. However, many of these algorithms are relatively inscrutable, and not fit for use by humans for casual game playing. Domain-specific languages (DSLs), used as the lexicon of techniques that evolve scripts for playing a game, offer the chance for more reasonable algorithms for understanding and usage by humans at the cost of some degree of complexity. However, how best to do this? One approach is to take one of the aforementioned more-complicated approaches and train an agent to use a DSL to approximate its actions. In this report, we use Monte Carlo Tree Search (MCTS) to prepare strong agents for the game *Can't Stop*, and then approximate their functionality with the Metropolis-Hastings algorithm on a context-free grammar using a DSL.

2 Background

2.1 Monte Carlo Tree Search

Monte Carlo Tree Search, or MCTS, is a method for determining the course of action in a game that involves using random playouts from nodes of the game tree to determine their value, then using those values to determine the proper course of action. It comprises 4 major steps: `Selection`, `Expansion`, `Simulation`, and `Backpropagation`.

At the `Selection` step, the game tree is traversed until a leaf node of the *unevaluated* portion of the tree is encountered; that is, at the `Selection` phase we check future game states until we encounter a game state that has no children for which we have performed a playout to determine its value. At the `Expansion` step, we select one of the children from the node we ended off on in the `Backpropagation` step. Next, we take that node and perform a random playout of it in the `Simulation` phase to get a value for it. Finally, we use that value to update previous nodes' estimated values in the `Backpropagation` step. The process is repeated until either some length of time has elapsed, or some number of simulations have played out.

2.2 Metropolis-Hastings Algorithm

The Metropolis-Hastings algorithm is used for drawing samples from high-dimensionality multivariate probability distributions[1] that would be otherwise hard to sample. It does so through

[1] while Metropolis-Hastings can be used for sampling lower-dimensional distributions, there are usually better techniques for doing so

usage of another distribution whose density is *proportional* to the original distribution.

To give an example, suppose we knew that some object followed a distribution $P : \mathbb{R}^2 \to [0, 1]$ on a 2D plane in such a way that it was more likely to show up along the boundary of a drawing of the Pokémon Ditto.

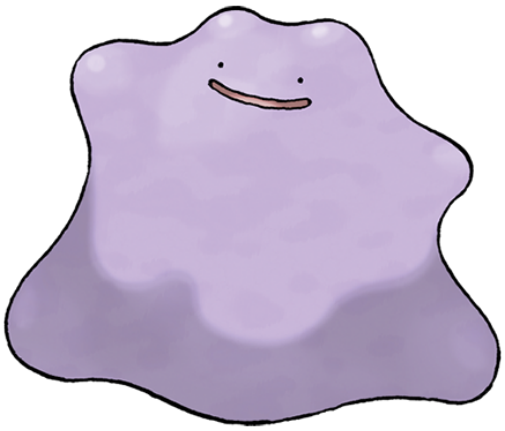

Figure 1: The Pokémon Ditto

It is not hard to imagine such distribution, and yet determining the normalizing coefficient is very much not trivial. However, by taking $f : \mathbb{R}^2 \to \mathbb{R}$ to be the distance of a point to Ditto's boundary, we have a function whose result is proportional the probability under the distribution. We then take advantage of the fact that therefore $f(x')/f(x) = P(x')/P(x)$ in the following routine:

1. Sample $x \sim \mathcal{N}((0, 0), \mathbf{\Sigma})$ to obtain some initial point in the plane ($\mathcal{N}$ here is sometimes referred to as the proposal function, as allows us to "propose" a new candidate given a center, as we will see)
2. Prepare a list of samples. While the number of samples in the list is lower than your desired sample count, perform the following:
   (a) Sample $x' \sim \mathcal{N}(x, \mathbf{\Sigma})$, so that x' is a proposed point "close" to x
   (b) Let α be the proportion $\frac{f(x')}{f(x)} = \frac{P(x')}{P(x)}$
   (c) Take a random value between 0 and 1. If your α is greater than this, add x' to your list of samples and let $x = x'$. If it is not, branch back to (a)

2.3 *Can't Stop* Board Game

Published by Parker Bros in 1980, *Can't Stop* is a turn-based dice-rolling strategy game where the goal is to advance tokens along columns. Rules for the game can be found at the above link.

3 Procedure

3.1 Input/Output Data Generation

To generate training data, we repeatedly play two UCT agents against each other to generate 10000 game-state/action pairs, where the game state is a state of the game within those matches and the action is the choice the UCT agent took. We allowed the UCT agents 150 iterations to ensure they were sufficiently strong.

3.2 Training Agents

To develop agents to mimic the UCT AIs' behavior, we use the Metropolis-Hastings algorithm mentioned earlier. Specifically, we take a DSL for *Can't Stop* and define a context-free grammar on it that builds up an abstract syntax tree to describe a script (see Figure 2 in subsection 4.2 for a visual example). In this manner, we have a space from which we can sample *Can't Stop* programs.

We start with our input/output data generated by the UCT agents. We then define a distance function $f : \{\text{Scripts}\} \to \mathbb{N}_0$ that returns the number of different output actions from the UCT agent the script would have taken over the inputs[2]. We define a sampling method that mutates a random near-terminal node in the abstract syntax tree representing a script as our proposal method, and have it occasionally modify more fundamental nodes further up the tree.

The top result from this (a script using a rule r_1) is then used to filter the input/output data. We repeat this Metropolis-Hastings application/filtering until we have 5 scripts (or "rules" as we will henceforth refer to them, as in actuality our we implemented AST to merely cover a subset of if-clauses).

4 Results

4.1 Found Rules

Below are the rules r_1, r_2, r_3, r_4, and r_5 gathered by the program. The initial sample count was 10000, generated by UCT Monte Carlo players picking their moves after 150 `Selection-Expansion-Simulation-Backpropagation` iterations.

4.1.1 Rule 1

```
if DSL.hasWonColumn(state,a) and DSL.isStopAction(a):
```

[2]This is directionally reversed from the distance function defined in the Metropolis-Hastings example in subsection 2.2, but this is accommodated for in algorithm

```
    return a
```

4.1.2 Rule 2

```
if DSL.containsNumber(a, 4):
    return a
```

4.1.3 Rule 3

```
if DSL.isStopAction(a) and 2!=0:
    return a
```

4.1.4 Rule 4

```
if 0==0 and DSL.containsNumber(a, 5):
    return a
```

4.1.5 Rule 5

```
if 2>=0 and DSL.containsNumber(a, 3):
    return a
```

4.2 Sample Abstract Syntax Tree

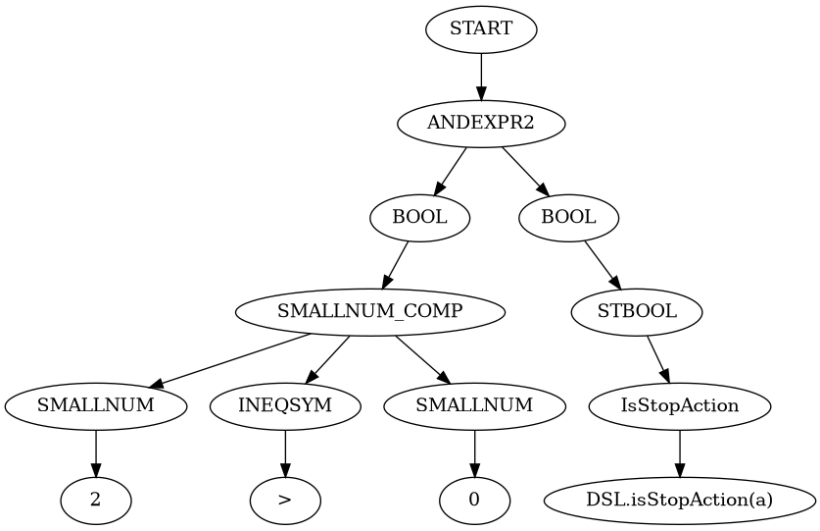

Figure 2: Abstract syntax tree for the expression `2>0 and DSL.isStopAction(a)`

5 Discussion

5.1 Relationship to Assignment 1

5.1.1 Rule Similarity

Rule r_1 (shown below in Figure 3) was identical to our results from Assignment 1.

```
if 2>=0 and DSL.containsNumber(a, 3):
    return a
```

Figure 3: Rule r_1, which also was the result of our first assignment

The other rules, however, were not. We suspect this to be because filtering on rule r_1 immediately biases the training/test dataset to be very different than what was prepared in Assignment 1.

5.1.2 Evolutionarily Stable Strategies and Nash Equilibria

Assume for a moment that a strategy S_1 solely utilizing rule r_1 forms an evolutionarily stable strategy in the context of Assignment 1. Then in the context of Assignment 1, S_1 does form a Nash equilibrium, as all ESSs are Nash equilibria.

This, however, does *not* mean that S_1 forms a Nash equilibrium for the smaller *Can't Stop* game; the context of Assignment 1 provides for a limited set of possible strategies in *Can't Stop*, and does not account for all possible mutant invaders of a population that could occur in the broader context of the game. What it instead means is that if we restrict the smaller version of *Can't Stop* such that the only legal behaviors are those given by our if-clause scripts, then S_1 is a Nash equilibrium for that hypothetical game.

5.1.3 Pros and Cons of Each Approach

Imitation learning and the genetic approach of Assignment 1 both of their upsides and pitfalls. Some of the pros/cons for each are given in Table 2 and Table 1.

Pros	Cons
Does not require an existing AI	Difficult to guide a GA's evolution
Any stable algorithms resulting likely able to stand up to a variety of mutant invaders (they are robust)	Algorithm still potentially vulnerable to unconsidered invaders
	Runtime grows with increases to the number of parameters open to mutation
	Contextually, GA evolution may be prone to getting stuck in local optima

Table 1: Pros and cons of a genetic approach

Pros	Cons
Development can be tailored to induce certain behaviors	Requires an existing AI or other trainer/-training samples
Learning can be modified to patch inefficiencies through adding more samples to the training data	Potentially does not generalize well to unseen patterns

Table 2: Pros and cons of imitation learning

5.2 Rules

I think rule r_1 provides useful information about being able to simplify the UCT's behavior, and the remaining rules speak to the nontriviality of the UCT. Given that rules $r_2 - r_5$ are generated in the absence of a large number of the UCT's decisions, though, I don't think they speak to the UCT's behavior directly. The rules found by the Metropolis-Hastings algorithm in the first iteration were all potentially useful, but I have reservations about the usefulness of the rules acquired in subsequent rounds by themselves.

Performance-wise, an agent using only rule r_1 performed fairly well (as mentioned, rule r_1 was the same rule as was generated in Assignment 1, so the agent's performance against other agents was the same). Inclusion of rules r_2-r_5 (contained in their own for-loops) to a script containing rule r_1 in its own for-loop resulted in agents that unfortunately performed worse against the r_1 agent, however.

One issue that all of the rules have in general is that the default action is baked into their associated scripts' functionality. As an example, an "always false" rule doesn't tell us anything about what a good action is, but the score for it will incorporate the result of selecting the default action. For future work, I would be interested in repeating the project, but when reaching the point of evaluating a rule's score I would provide penalty both whenever a rule picked a wrong action *and* whenever said rule failed to pick an action.

6 Code

All code for this project can be found at `https://github.com/dem1995/imitation-learning`