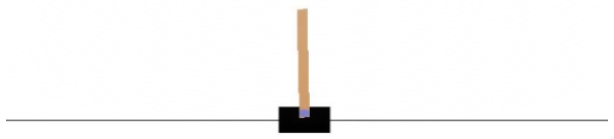
Independent Study

Report-4, 15th March

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

Simpler Problem first

While working on the program synthesis code, we realized that the Breakout environment might be a little too complicated for us to begin with due to the vast number of features and the fact that the oracle itself was so hard to train in the first place. So we decided to solve things for the well-known and much easier Cartpole Environment first.



Cartpole Oracle

Instead of using a DQN, we decided to move forward with a simple tabular Monte Carlo algorithm for our oracle. The training occurs fairly quickly with a surprisingly decent performing model in the end. At the end of the training, the model is nothing but a table containing the action to be taken for a given state, since the states lie on a continuous 4D space, the actual environment observation is first converted to a discretized state and then then the action can be found by doing a simple lookup on the table.

Program Sketch

We've chosen a very simple sketch as shown in the antlr4 CFG below, where the Non-Terminal e represents our program, output '0' means to move left and output '1' means to move right.

```
e : '0' | '1' | IF b THEN e ELSE e;
b : cond OR b | cond AND b | cond;
cond : term ( '+' term )* '>' INT;
term : INT '*' peek;
```

```
peek : PEEKTKN '(' H ','  INT ')';

/*Tokens*/
IF : 'if';
THEN : 'then';
ELSE : 'else';
OR : 'or';
AND : 'and';
PEEKTKN : 'peek';
H : 'h0' | 'h1' | 'h2' | 'h3' ;
ACTION : '0' | '1' ;
NS : [ \t\n]+ -> skip;
INT : ('-')?[0-9]+;
```

Here, H basically represents the history of observations from one of the sensors and since there are only four observations in the Cartpole Environment, we have accordingly h0, h1, h2, and h3. Rest all symbols have their usual semantics.

An example program from the above CFG:

```
if peek(h0, 2) + peek(h2, 3) > 0 and peek(h1, -1) > 7
then
      0
else
      1
```

Program Initialization

We first initialise a random set of programs, which will form our initial search pool.
Our algorithm expands Non-Terminal symbols randomly, till the parse tree becomes equal to a target depth in size. We ensure that the leaves of the final parse tree are terminals. The leaves, taken in order, form one PiRL program.

The algo for program initialisation is as follows:

```
INPUT: depth, grammar_rules, start_symbol
```

```
function CheckRuleDepth(rule, depth):
      // checks if there exists some valid set of expansions that
      // will produce a tree of depth = depth, starting from the
      // current rule

      if depth == 1:
            if all symbols in rhs of rule are terminals:
                  return True
            return False

      for each symbol in rule RHS:
            if symbol is non terminal:
                  valid_next_rule = False
                  foreach next_rule in (rules starting from symbol):
                        valid_next_rule |= checkRuleDepth(next_rule, depth-1)

                  if not valid_next_rule:
                        return False
      return True


function generate(start_symbol, depth):
      // randomly generates a tree of depth = depth starting
      // from the start_symbol

      valid_rules = []

      foreach rule in (rules starting from start_symbol):
            if checkRuleDepth(rule, depth):
                  valid_rules <- valid_rules + rule

      if len(valid_rule) == 0:
            throw "No tree of depth {depth} possible"

      chosen_rule = random.choice(valid_rule)
      node = Node(start_symbol)

      foreach symbol in rhs of chosen_rule:
            child = Node(symbol)
            if symbol is non terminal:
                  child = generate(symbol, depth-1)

            node.children.append(child)

      return node
```

Optimal Parameter Search

When generating a new program, in place of unknown numerical constants, we generate a special symbol. This allows us to restrict the search space by removing the generation of unknown numeric constants.

We can parametrize each program, using these unknown numerical constants. We then perform a regression based optimisation, on the values of these numerical constants, to obtain the program performing closest to the oracle.

Neighbourhood Pool

To generate neighbours:

1. we take the best generated program or simply the current iteration program template
2. replace the constants with the special symbol for numerical constants
3. remove a subtree of the parse tree, and regenerate that subtree, while giving priority to the shortest program

The programs thus generated form the neighbourhood pool.

Future Work

We plan to complete the code for Neighbourhood Pooling and Program Initialization for the next phase. Since, we also have the quiz week coming up, we might not be able to start with the Optimal Parameter Search part till the next phase.