Independent Study

Report-5, 1st April

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

Program Generation

We first needed to figure out how to generate programs according to the given sketch grammar. The steps for this are as follows:

1. Parse the sketch grammar using antlr.
2. Traverse the tree and convert into a set of rules.
3. Start with the start symbol, and randomly choose a “valid expansion”.
    a. An expansion (*b)* is called “valid” if there is some set of productions such that the final string has only terminals, i.e. there is at least 1 possibly parse tree which ends in all terminals.
4. This allows us to construct a parse tree. We can take the leaves of this parse tree to generate the actual program, but for our use case, we just require the parse tree.

Having generated the program parse tree, we need to convert this into an optimisation task, so that we are able to use standard optimisation techniques like gradient descent to figure out the values for the constants.

For this, we have defined some “standard constructs”. These include:

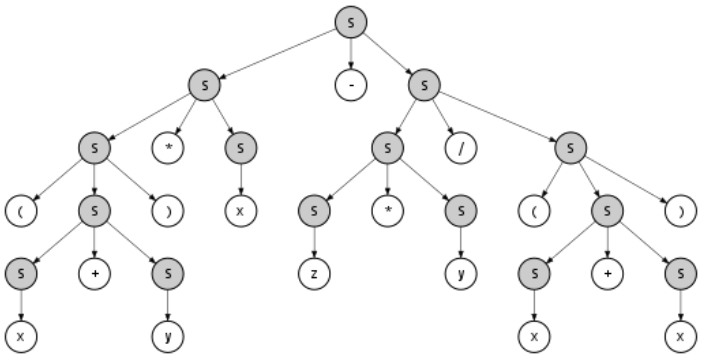
1. if condition
2. filter : functional programming filter operation
3. map : functional programming map operation
4. peek : returns the value of the last observation in history
5. binary operation : +, -, *, /
6. return statement
7. input variables
8. constants

We map these constructs to tensorflow programs. The aim is to optimise for constant. The loss function is the cross entropy between the value returned by our program and that determined by the oracle.

This step gives us the most optimised version of a given program. In further steps, we will evaluate the cost of this, most optimised program, and compare that with neighbouring programs to figure out which program to pick next.

Neighbourhood Pool

To find better program templates iteratively, we explore the program space by generating a pool of the current best program's neighbours. Generating a given program's neighbours is fairly easy if we have the parse tree, we simply need to replace any random Non-Terminal subtree in the parse tree with another valid subtree.



A simple parse tree with shaded nodes representing Non-Terminals

```
// Pseudocode
1. generatePool(numOfNeighbours, maxDepth, parseTree)
2.     NTNodes = [Non-terminal nodes of parseTree]
3.     neighbours = []
4.     for i in 1..numOfNeighbours
5.           treeCopy = deepCopy(parseTree)
6.           randomNTNode = select random element from NTNodes
7.           newSubTree = generate appropriate depth subtree from randomNTNode
8.           replace subtree with randomNTNode as root with newSubTree in treeCopy
9.           neighbours.append(treeCopy)
10.    return neighbours
```

For Example, lets taken the given program:

```
if c*peek(h0, -1) + c*peek(h1, -1) + c*peek(h2, -1) + c*peek(h3, -1) > c
then
    left
else
    left
```

The 5 generated neighbours are as follows:

```
# Neighbour-1
if c*peek(h0, -1)+c*peek(h1, -1)+c*peek(h2, -1)+c*peek(h3, -1) > c and
   c*peek(h0, -1)+c*peek(h1, -1)+c*peek(h2, -1)+c*peek(h3, -1) > c or
```

```
   c*peek(h0, -1)+c*peek(h1, -1)+c*peek(h2, -1)+c*peek(h3, -1) > c and
   c*peek(h0, -1)+c*peek(h1, -1)+c*peek(h2, -1)+c*peek(h3, -1) > c
then
    right
else
    if c*peek(h0, -1)+c*peek(h1, -1)+c*peek(h2, -1)+c*peek(h3, -1) > c
    then
        left
    else
        right

# Neighbour-2
if c*peek(h0, -1)+c*peek(h1, -1)+c*peek(h2, -1)+c*peek(h3, -1) > c and
   c*peek(h0, -1)+c*peek(h1, -1)+c*peek(h2, -1)+c*peek(h3, -1) > c or
   c*peek(h0, -1)+c*peek(h1, -1)+c*peek(h2, -1)+c*peek(h3, -1) > c
then
    left
else
    right

# Neighbour-3
left

# Neighbour-4
if c*peek(h0, -1)+c*peek(h1, -1)+c*peek(h2, -1)+c*peek(h3, -1) > c
then
    left
else
    if c*peek(h0, -1)+c*peek(h1, -1)+c*peek(h2, -1)+c*peek(h3, -1) > c and
    c*peek(h0, -1)+c*peek(h1, -1)+c*peek(h2, -1)+c*peek(h3, -1) > c
    then
        if c*peek(h0, -1)+c*peek(h1, -1)+c*peek(h2, -1)+c*peek(h3, -1) > c
        then
            right
        else
            right
    else
        if c*peek(h0, -1)+c*peek(h1, -1)+c*peek(h2, -1)+c*peek(h3, -1) > c
        then
            right
        else
            left
```

```
# Neighbour-5
if c*peek(h0, -1)+c*peek(h1, -1)+c*peek(h2, -1)+c*peek(h3, -1) > c
then
    left
else
    left
```

What Next?

We have completed the code for Program Generation and Neighbourhood Pooling. Next, we plan to write code which optimises the parameters for a given program template as described in the Program Generation section using Tensorflow backend.