Learning to Build Towers via a Compositional Language

Kevin Ellis, 6.868 Final Project

1 Overview

In this project, I propose and implement a model of how one could learn and acquire new critics and selectors. I apply this model to the problem of learning to build tall, stable towers of blocks in a simulated physical world. My program is capable of building new critics and selectors from past experience, and of applying these agents to solve new, harder problems. For example, it is able to solve harder block building problems after it has solved easier ones.

In the spirit of Sussman’s PhD thesis [1], I chose to represent procedural knowledge about how to build towers in the form of programs. My system uses programs expressed in combinatory logic, which is a Turing-complete functional programming language.1 Currently the system learns “Selectors” that propose changes to the current block tower; a framework also exists for defining “Critics” that point out flaws in the current design, which has not yet been implemented.

In addition to learning Selectors (and, once implemented, Critics), the system also learns a library of program fragments that it composes when trying to write new programs. Combinatory logic is an exceptionally compositional representation, and so lends itself well to this sort of library. By building a library of useful program pieces garnered from past experience, the system can tackle harder problems once it has solved easier ones.

The main ideas motivating this system are:

“You don’t understand anything until you learn it more than one way” -Marvin Minsky.
In the first implementation, the system would build a tower once and then move on to the next building challenge; I found that, in doing so, it was rarely able to transfer lessons learned from the easier tower to the harder one. So, I instead programmed it to try solving each problem in many different ways, and found that only then could it transfer skills from easy challenges to hard ones.

“You can only learn what you almost already know” -Patrick Winston. A complicated tower-building challenge - one where the tower must be tall, or where it must stay upright when shook - cannot be solved by a tower-naive individual. It is only after a graded curriculum of easier challenges that one is able to solve difficult problems.

Our brains contain world-simulators that allow us to reason about physics.[3, sec. 5-5] Clearly people have some sort of intuitive knowledge of physics that allows them to design ad-hoc structures, make stability judgments, etc. The system emulates this

1Note that combinatory logic is not a logic.
ability by using a physics engine designed for computer games, and then adding random forces to simulate both uncertainties and a tower being shaken. Although this simulator is undoubtedly more accurate than whatever “physics engine” lies within our skulls, the addition of random forces might justify its use as a first approximation to the physics machinery of the brain.

Compositional representations encourage transfer learning. The system represents agents (so far, only selectors) as arbitrary computer programs. The language used is compositional: every program is a tree, and every subtree of a valid program is always a valid program. Any two programs can be combined via function application. This compositional nature permits the reuse of parts of programs, and allows the learning of higher-order concepts, such as “Take any selector, run it, and then use only its first suggestion,” or, “Reflect the plan of this other selector across the axis specified by some other program,” etc. Programs parts found useful in certain contexts are learned to be reused in similar contexts.

2 Implementation

2.1 Incubation

Incubation is the process of enumerating a large number of potential solutions to a given problem [3, 7-7]. In my system, solutions are given by programs, and incubation is a recursive, partly random process which I will now describe:

Incubation is given a library of program fragments, each with a weight. It is also given a target type, which is the type of the program to be written. This can be monomorphic type, such as Int, Bool, or Int -> Bool (functions from integers to booleans). Alternatively, the type can be polymorphic, such as \( \alpha \rightarrow \alpha \) (all functions that return data of the same type that they receive). Incubation randomly chooses to either sample the library, in which case it draws a program fragment from the library of the correct type with probability proportional to the fragment’s weight, or it can choose to recurse, in which case it incubates a function and an argument, and then applies them. The implementation performs Hindley-Milner type inference as it incubates programs, ensuring that the resulting expression is well-typed.

2.2 Planning (Deliberation)

Planning is done with the assistance of a set of selectors, obtained via the incubation process. Each selector is a program that takes as input a plan, and produces as output another plan. plans are lists of blocks, specified as a tuples of location and orientation. The planner runs each selector on the current plan, producing a set of proposed new plans. In future implementations, I intend to have the plan chosen to be a function of a set of critics, which are programs that, given the state of the world, evaluate to a real number. This number could be higher for states that aren’t worth pursuing; alternatively, one could imagine more sophisticated critics that instead assigned selectors to states. However, the current implementation just picks the next state randomly, with each state being equiprobable.
2.3 Reflection
The program remembers all of its attempted tower constructions; it analyzes them, and assigns credit to the selectors that were essential to the construction of good towers. Then, it updates the library as follows:

First, it finds the selectors with the most credit, and attempts to partition them such that any two selectors in the same partition roughly represent the same “way of thinking.” It does so by running the high-credit selectors on various initial plans, creating a “fingerprint” for each selector. Any two selectors with the same “fingerprint” are considered to represent the same way of thinking. Credit is then equally distributed among the various ways of thinking, so that, for example, a slightly less useful selector that is dramatically different from the other selectors will nevertheless be given high credit.

Now that credit has been assigned to programs, the library is synthesized by incorporating all of the selectors into it with weight proportional to their credit. Additionally, common subparts of selectors are factored out and added as fragments to the library.

I also tried synthesizing the library by treating it as a grammar estimation problem, and adapting existing algorithms for inducing probabilistic context free grammars. I found that this approach led to the program getting stuck in local minima, both in library space, and in the planner’s space. The code for performing this grammar induction is saved in Reflection.hs. I chose not to use it because it hurt performance, and is unmotivated from a Society-of-Mind perspective. It also cannot result in learning more than one way of thinking, a property I found to be essential in the experiments.

2.4 The Code
The system is implemented primarily in Haskell with a few Python utilities. The Haskell code is responsible for managing combinatory logic expressions (CL.hs, BlockLib.hs, Pack.hs), incubation (Incubation.hs), reflecting on past constructions making new libraries (Reflection.hs), and planning (CriticSelector.hs). It interfaces with the Python scripts in one of two ways: either over a network, for a remote host responsible for running the physics simulations, or locally, if the machine running the Haskell code is also responsible for the physics simulation. The networking code is in BuilderClient.hs, and a caching system that greatly speeds up the physics simulation is in PhysicsCache.hs. The Haskell application is parallelized and runs much faster on multicore machines.

There are three Python applications, server.py, render.py, and console.py, all located under the simulator directory. server.py takes no arguments and starts up a physics simulator server on port 1540. console.py is a utility that is invoked by the Haskell application when it wants to run local physics simulations. render.py is a utility that allows one to render and simulate towers in the block world. It takes two arguments, the first of which is a list of tuples of numbers and booleans; each number is a horizontal coordinate, and each boolean is True, for horizontal blocks, or False, for vertical blocks. The second argument is a positive real number, corresponding to the strength with which the tower will be shook. For example, python render.py "[(-1,False),(1,False),(0,True)]" 1.5 will build an arch and then try to knock it down by shaking it. Code common to all of these scripts is located in builder_common.py. The Python scripts use the Panda3D game engine and the
Bullet physics engine, both of which must be installed in order for the system to work.

3 Experiment

I tested the system’s ability to learn how to solve a hard building task under three different regimes:

“Transfer” regime: The system is first presented with an easy building task. It has to solve the task many (30) times, ensuring it learns different ways of solving the same problem. It is then presented with the hard task. In this regime, it successfully accomplished the hard task 27/30 times.

“No Transfer” regime: The system is immediately presented with the hard task. No transfer of knowledge (selectors and program fragments) is permitted from previous problems. In this regime, it only accomplished the task 9/30 times.

“Bad Transfer” regime: The system is first presented with an easy building task, but moves on once it has found a solution. It is then presented with the hard task. In this regime, it did even worse than with no transfer learning: it only accomplished the hard task 5/30 times. In observing the towers it was trying to build, I noticed that it was “stuck” in thinking that the best way to approach the problem was using the one “way of thinking” it had used on the easy task.

The easy task was to build a “tower” of at least one block, with very little shaking of the tower. One example solution to the task, found by my program, is shown in Fig. (1). The hard task was to build a tower using no more than three blocks, with moderate shaking of the tower. The tower had to be of moderate height. The actual numerical values of height and shaking strength are irrelevant, because their units don’t correspond to anything physical. They are given in the code in the file Reflection.hs. One possible solution to the hard task is an arch, which was found by my program. It is shown in Fig. (2).

Figure 1: Example solution to the easy task
4 Conclusion

The project illustrates an important lesson I learned from Society of Mind and Emotion Machine: that artificial intelligences need multiple means of solving problems in order to not get stuck. Only by solving an easier problem in multiple ways was my program able to tackle a harder problem.

Human beings are able to acquire a large number of ways of thinking, and are also able to transfer solutions from one problem on to another, more difficult one. I hypothesize that the wide range of ways of thinking can be explained by representing new mental resources as arbitrary computer programs, coded in some mental programming language. Our ability to synthesize new ways of thinking from old ones indicates that this language must be compositional; in artificial intelligence programs, combinatory logic, or something similar to it (lambda calculus, etc), is a convenient model.

In future extensions, I intend to experiment more with this system, as well as add the ability to learn critics in a manner similar to how selectors are learned. I am also interested in extending the planner to use the bidirectional search ideas described in [3, sec 5-3].

References

