Description of the agent

Susumu Katayama

August 14, 2017

1 The principle

The challenge can be formalized as the problem of adaptively synthesizing the agent's behavior at each time step as a function from the interaction history to an action. Our main idea is to synthesize the function as a program which uses a finite set of library functions (including nullary values) or *skills*. Incremental learning can be achieved by biasing the library to fitter, more useful functions (or functions appearing in useful compound functions more frequently). *Biasing* means searching promising regions deeper, by using more complex compound library functions. We call the functions in the initial library *instincts*, and we call the useful set of functions that have been synthesized as combinations of instincts and registered to the library *learned skills*.

1.1 The learning cycle of our agent

Figure 1 roughly shows the learning cycle of our agent.

The learner tries to detect task instance switch, and at this timing, it updates the library (or set of skills). In other words, one cycle corresponds to one task instance.

One cycle has two phases:

1. the observation phase during which the agent only randomly pretends to communicate with the environment and collect the history information, and

2. the main phase during which it truly communicates intelligently.

After the observation phase, lazy stream of all programs that can be composed from the set of skills and are consistent with the history so far is generated. This part is implemented using the MagicHaskeller library. Each program has an associative memory (that maps input strings to feedbacks) and conditional reflex (that maps inputs to outputs) generated from the history.

In the main phase, the agent applies programs in the stream to the current input character and history, and remove programs that are apparently inconsistent with the input and the history. Programs should not be removed if looking up their associative memory fails, because they may become consistent when the associative memory is substantiated later.

Then, the algorithm finds the leftmost consistent program and its return value, and it is used as the output character to the environment. In this algorithm, the leftmost program is the one with the highest priority. At the beginning, it is the shortest valid program. When a program's return value caused to receive positive reward, it is moved to the leftmost position.

Then, the history is updated, the algorithm repeats the main phase from the beginning, unless it detects a task instance switch. How to detect a task instance switch is discussed in the next section.

Then, the set of skills is updated, by adding subexpressions of the last rewarded synthesized program.

1.2 Detecting task instance switches

Incremental learning is achieved by updating the library, or the set of skill functions. It is updated when the algorithm detects a task instance switch.

We think that correctly detecting them is a difficult task, if ever possible, even for humans without prior knowledge or non-verbal signs, though overlooking switches can cause catastrophic results. Thus, we decided not to spend lots of time for this, and assume a switch when the current behavior that has been doing well suddenly start being punished or failing to parse the history.

1.3 Choice of the set of instincts

Practically, an important question is how the set of instincts looks. There is a trade-off between generality of efficiency of the learner: if all the tasks are known in advance, the agent can learn in the least number of steps by using a set of complex compound functions which are specialized to the task; on the other hand, the agent can deal with more different tasks by increasing the percentage of primitive instinct functions.

On applying the quantitative prize, the learner must solve more tasks in less time within 24 hours. Thus, we made a guess on the extent of the required generality, and designed an "animal-like" (rather than human-like) set of instincts which includes compound functions. Apart from this choice of the set of instincts (which has been solved by evolution in the case of animals in millions of years), we believe that our algorithm is designed in a very general framework and can be applied to design of general AI.

A How to compile and run the agent

A.1 Compiling the agent

We assume Ubuntu 14.04 or 16.04. Install Haskell Platform and 0MQ-related packages by

```
> sudo apt-get update
> sudo apt-get install haskell-platform \
      libghc-zeromq4-haskell
```

Now create and move into a new directory. Then, typing

```
> unzip source.zip
> cabal update
> cabal install
```

will install executable `ZMQAgent` into `./dist/build/ZMQAgent/` and `~/.cabal/bin/`.

The default build of `ZMQAgent` accesses `tcp://localhost:5556` and does not log the history.

```
> cabal install --flags="DOCKER"
```

changes the target address to `172.18.0.1`.

```
> cabal install --flags="MONITOR"
```

makes the executable which writes out input-output-reward history to the standard output.

If you prefer to choose another target address, e.g. `1.2.3.4`, execute

```
> ghc ZMQMain.hs -package zeromq4-haskell \
    --make -O2 -DADDRESS=1.2.3.4
```

then, `./ZMQMain` will be generated as the executable.

A.2 Running the agent

`ZMQAgent` (and `ZMQMain`) can be run without arguments, e.g.,

```
> ./ZMQAgent
```

or from within the `Round1` directory

```
> python src/run.py \
   src/tasks_config.challenge.json \
   -l learners.base.RemoteLearner \
   --learner-cmd "path/to/ZMQAgent"
```

If command line arguments are provided, the first one is used as the port number and the rest is discarded.

```
skills <- instincts

forever {

  history  <- []          // empty list
  alphabet <- []

  for a while {
    // just observe without synthesizing anything
    if input==' ' then output random char
                  else output ' '
    history <- history ++ [(input,output,reward)]
    alphabet <- alphabet ++ [input]
  }

  progs <- all the valid programs composed of
           skills and alphabet,
           from the shortest increasing the length

  do {
    progs <-
         [ prog | prog <- progs,
                  prog(input) is not an error
                  (i.e. prog is consistent)
                  or assoc memory lookup failed ]

    candidates <- [ (prog, prog(input)) |
                    prog <- progs,
                    prog(input) is not an error ]

    (candidate, output) <-
                  initial element of candidate

    if reward>0
       then progs <-
          [candidate] ++ progs without candidate

    history <- history ++ [(input,output,reward)]
  } until a task instance switch is detected

  // Increment the skills!
  skills <-
    skills ++ subexpressions(synthesized_program)
}
```

Figure 1: A sketch of the learning cycle of our agent. In this pseudocode, `++` is the list-concatenation operator, and `[ . | . ]` denotes list comprehension, that is the list counterpart of set comprehension.