General AI Challenge submission

Andreas Ipp (August 14, 2017)

Instructions

To launch the agent, do the following:

```
python3 agent/example_learner.py [-a <address> -p <port> -i]
```

The option **-i** launches the agent in **interactive mode**. Entering 'help' shows available commands, like evolving the system for some time and inspecting its internal state.

Design of the agent

The agent guesses possible solutions to the tasks, which are represented as mini-programs stored in a "**genetic code**". Best candidate solutions are tried and selected, and new solutions are obtained by various genetic transformations (mutation, crossover, injection of genes at particular sites, …). **Gradual learning** is achieved by storing successful "genes" and starting with those when a new task starts.

On the following pages, it is argued that a **general solution** to a general CommAI-Env challenge (with arbitrary input and reply bit or byte streams) **can not exist**, as the history of teacher's input bytes and student's reply bytes does in general not restrict the future reward function at all - it can in principle be anything, even after a very long history.

Therefore, we can only try to solve a particular implementation of the CommAI-Env - which is the one from this challenge. In this case, the **restriction of the reward function** that we search comes from the particular choice of **task switches** (at least 10 positive replies in a row) and the restriction of the reply / reward function to **simple Python programs**.

Closed source

For now, I want to share the agent's code only with the organizers.
The agent is a single-threaded Python application and does not use the GPU.

Main principles and motivations

My main observation (I don't know if this is obvious to everyone) is that **the general CommAI-Env challenge is impossible to solve** - not even in principle.

This can be seen as follows:

> We assume a deterministic teacher (for the same teacher's input history and the same student's reply history, the same correct student's reply is expected), but we **assume that the student has only 1 instance** (that is "one life") of solving the tasks - as is the case in this General AI Competition. Then the reward function $R \rightarrow \{-1, 0, 1\}$ can in principle depend on the whole history of n teacher input bytes $i_1, i_2, i_3, \ldots i_n$ and n student reply bytes $j_1, j_2, j_3, \ldots j_n$. What we **learn** is a **series of reward functions**:
>
> $R(i_1; j_1) \rightarrow r_1$ (1 out of 256^2 possible functions)
>
> $R(i_1, i_2; j_1, j_2) \rightarrow r_2$, (1 out of 256^4 possible functions)
>
> ...
>
> $R(i_1, i_2, \ldots i_n; j_1, j_2, \ldots j_n) \rightarrow r_n$, (1 out of 256^{2n} possible functions)
>
> which **tells us absolutely nothing about the next reward function** after n+1 bytes:
>
> $R(i_1, i_2, \ldots i_n, i_{n+1}; j_1, j_2, \ldots j_n, j_{n+1}) \rightarrow r_{n+1}$, ($256^{2n+2}$ possible functions)

Note that if you have "several lives" or attempts for interacting with exactly the same teacher from the beginning, then it is easy to map out all reward functions. The point is, if there is only one try, it is absolutely impossible - unless there are further assumptions that can be made about the teacher.

One such assumption can be that the reward function only depends on the **recent history**, for example only on the last m bytes of input and reply with m being constant. Then, given enough time, it is possible to map out the whole reward function. But this assumption **precludes the possibility of long-term memory** (for example, the teacher defining something at the beginning of the history, and referring to it later after >m bytes).

But even if we restrict the reward function to a recent history of m bytes, the challenge can still be **impossible to solve** if the teacher has some **additional internal state** s that we don't know about. (Simple example: the teacher can be in one of two states - in the first state, the correct answer is "a", in the second state, the correct answer is "b". If the teacher switches between these two states randomly from reply to reply, the student just can not know the correct answer.)

Given that a general solution to the general CommAI-Env challenge does not exist, this actually takes a large burden, as there is no point in trying to look for such a general solution. The best we can do is to find a solution for the particular implementation of the teacher. Any solution to the CommAI-Env challenge therefore has to make **assumptions** about the internal workings of the teacher. In this case, the assumptions are:

- The **internal state** of the teacher is defined by one of several possible **tasks**. (In different tasks, the respective correct answers can differ). The teacher stays within the same task until a certain number of positive rewards is given. This defines a task switch.
- The solutions to the tasks can be expressed as **simple Python snippets** - as is evident from the source code.

Therefore the **solution strategy** for solving this task under the assumptions of the given implementation of the CommAI-Env is the following:

- Within a task (which starts after a series of at least 10 positive rewards ends), try to **find a short Python program** that is consistent with the observed inputs, replies and rewards.
- The following pseudo-algorithm can in principle solve the task:
    - Write down all possible Python programs (sorted by increasing length and complexity), which take the history within the current task as input and give a reply byte as output, and **select the first program that gives consistent output for all replies within the current task** (since the last task switch) - that is: the reply for positive rewards should be identical, and for negative rewards should be different.
    - Use the program to calculate the next reply byte. If the reward is negative, continue the search (using from memory the whole history of the known part of the reward function since the last task switch)

For practical purposes, the number of even very short Python programs can be huge such that an exhaustive search is impractical. Furthermore, note that being **Turing complete**, in principle all reward functions R can be generated, leaving no predictive power at all (one can easily write code that gives exactly the same output for the known n bytes, but different output at the n+1 byte position). The **restriction of the reward function *R*** comes from the fact that not all reward functions can be generated from programs of limited length and complexity.

Conversely, **given any agent, one can find a new implementation of the teacher that the given agent can not solve**. Here are two simple examples, that probably none of the agents of this competition can solve:

- The teacher's n-th input is the $(2n-1)$-th digit of π=3.14159265358..., and the student should reply with the $2n$-th digit: 3→1, 4→1, 5→9, 2→6, 5→3, 5→8, etc.
- Any reply is correct if the student waits for at least 1 second. (Very easy to solve for humans!)

Both solutions have actually simple Python programs, and once it is known what the teacher wants (or what the internal assumptions are), it is easy to adapt the agent to such a solution.

Implications for the “big picture”

In the spirit of the General AI Challenge competition, let me try to draw some conclusions:

- Given that no solution exists for the general CommAI-Env challenge, and the solution to the particular implementation of the CommAI-Env challenge consists in principle of trying to find short Python programs, it is **questionable that the CommAI-Env can bring us much closer to the quest of general artificial intelligence**.
- If we want to shape artificial agents that **act as humans** would do, the teacher should actually resemble the environment of the human.
- It may be surprising, but the result may indicate that there can not even be a **superintelligence** that can solve the general CommAI-Env challenge, since a general solution does not exist. It can only solve it under given assumptions - and if the assumption is to act as human-like as possible, that is a possible goal. But then the question is where the superintelligence would evolve to from that point on. In any case it will be **limited by its underlying assumptions** - which will be derived from those that humans have (and without these assumptions it might lose its predictive power).
- As usual, there can be many loopholes in such an argumentation and the future may be totally different from its prediction (as in the general CommAI-Env).

Appendix

Implementation and its limitations

In principle one could generate random short strings and apply Python's eval() function. It was easier to start with a simple stack machine, where one has full control over execution time, memory usage, etc.

- As long as a solution is representable by the genetic code, there is a chance of finding it, but it may take a long (random) amount of time.
- So far only a very limited set of Python commands has been implemented in the "genetic code". If other functions are required in unknown tasks, the agent may not be able to solve those.
- The current implementation of the agent knows 39 OP-codes (Jump, If, Push, Pop, …), with a stack that holds the teacher's input before execution and the student's output after execution.
- A random access memory is implemented in the form of a Python dictionary, to which the genes can write to and read from with particular OP-codes.
- Some string operations within the dictionary are directly accessible by OP codes with parameters provided through the stack (for example take or delete part of strings, join or split strings, etc.)
- 42 gene operations are defined - besides common operations (random mutation, crossover, appending, …), particular operations are indicated by gene annotations - for example to easily replace parts that act on strings
- The agent can at times get stuck at a wrong set of solutions for a long time - therefore the times for finding the correct solution may vary a lot from run to run.
- Various levels of "panic mode" are implemented in order to avoid getting stuck: if no solution is found, random characters are tried, first from the set of characters used by the teacher, then by all characters; the genetic pool is reset if no solution can be found.
- In order to maintain a diverse genetic pool, genes that have different genetic codes but same output are restricted to a maximum number.
- The fitness function defines several possible states, depending on the own reply of a gene, the overall reply that was given, and the reward by the teacher. Besides "right" or "wrong", it can also be "possibly right" (different answer than the wrong answer) or "probably wrong" (different than the correct reply).
- The fitness function also contains the number of right predictions that a gene made - as well as the number of tasks it survived.

Sample genetic code

Here is some sample genetic code that is used in solving the **first micro-tasks**:

This code takes one byte from the input stack, and replaces it by another byte, if a certain value is found.

```
genes = [
    3, var1, 3,    # if stack contains 'var1'; otherwise jump +3
    2, var2,       # replace by 'var2'
    5,             # break
    0              # drop stack
]
```

The first column is the OP code (for example “3” == IF, “2” == Replace stack entry, etc.) and the following items are arguments to the functions.

For the **micro-tasks 5 onward**, an elaborate protocol with the teacher is used, including internal states when to speak and when to listen. All of this can be expressed with the “genetic code”, but it was quicker to code this section by hand than to wait for it to become evolved automatically.
Code annotations at the end indicate, at what positions further code can be injected or replaced. (Side note: In nature, such kind of replacements are realized by the CRISPR/Cas9 genome editing tool)

The following **sample code** consists of 152 bytes, and is responsible for handling question / answer pairs with input and feedback separators:

```
genes = [              # Stack: [c]
                       # Fixed dictionary constants:
                       #   0: Mode:
                       #        0: continue to read input (until '.')
                       #        1: wait for feedback quietly (while ' ')
                       #        2: continue to read feedback (until ';')
                       #        3: wait for feedback pushing separators (while ' ')
                       #   1: Input string
                       #   2: Output string
                       #   3: Copy of input string for processing
                       #
       3, var2, 14,    # If stack == var2:  [c]   <-- feedback separator ';'
       0,              #   Drop stack (var2) []
                       #
                       #   | <-- Replace code (4:+5) [store_input_feedback_pair]
                       #   |
                       #   | Store input-feedback pair in dictionary
                       #   | (optionally: remove first part of feedback)
                       #   | (optionally: drop if last character of feedback == '.')
```

```
1, 2,           #   | Push 2 [2]
1, 1,           #   | Push 1 [1, 2]
14,             #   | Put string into dictionary [] {[1]: [2]}
                #
                #   Change mode to 0
1, 0,           #   Push 0 [0]
1, 0,           #   Push 0 [0, 0]
8,              #   Put into dictionary [], {0: 0}
                #
1, var1,        #   Push var1 (reply separator) [' ']
5,              #   Break [' ']
                #
3, var3, (18    # Elif stack == var3:  [c]   <-- input separator '.'
 +13+14+18+10), #
                #
                #   | Optional part: if mode != 0, continue below
1, 0,           #   | Push 0 [0, c]
10,             #   | Is in dictionary 0 [result, c]
3, 1, 6+14,     #   | If result == 1: [1, c]
2, 0,           #   |   Replace 0 [0, c]
9,              #   |   Get from dictionary [mode, c], {0: mode}
                #   |
                #   |  | <-- Replace code [store_input_separator_in_reply]
                #   |  |
                #   |  | Optional part: If mode != 0, don't store '.', but reply
4, 0, 11,       #   |  | If mode != 0: [mode, c]  <-- read input mode
1, var7,        #   |  |   Push 2 [var7, mode, c]
3, 1, 5,        #   |  |   If stack == 1: [var7, mode, c]
0,              #   |  |     Drop stack [mode, c]
0,              #   |  |     Drop stack [c]
2, var1,        #   |  |     Replace ' ' [' ']
5,              #   |  |     Break [' ']
0,              #   |  |   Drop stack [mode, c]
                #   |
3, 0, 30+18+10, #   |   If mode == 0: [0, c]  <-- read input mode
                #   |     continue, otherwise jump to [CONTINUE] below
0,              #   | Drop stack [c]
                #
                #   | <-- Replace code [store_input_separator]
                #   |
0,              #   | Drop stack []
                #
                #   <-- Inject code here for starting own reply [start_reply]
                #
                #   Change mode to 1
1, 1,           #   Push 1 [1]
1, 0,           #   Push 0 [0, 1]
8,              #   Put into dictionary [], {0: 1}
                #
                #   Copy dictionary item from 1 into 3
1, 3,           #   Push 3 [3]
1, 1,           #   Push 1 [1,3] {1: 'abc'}
11,             #   Copy dictionary item [], {1: 'abc', 3: 'abc'}
                #
                #   Get string from dictionary [3] [lookup_reply]
1, 3,           #   Push 3 [3] {3: 'abc', 'abc': 'def'}
15,             #   Get string from dictionary [], {3: 'def', 'abc': 'def'}
```

```
                #
                #   Optional post-processing of data:
                #   <-- Inject code for string manipulation (61)
                #
                #   | <-- Optionally remove code [reply_separator]
                #   |
                #   | Optionally Append '.' to reply (18 "bytes")
                #   | (reply only if answer non-empty)
1, var6,        #   | Push 2 [var6]
3, 1, 12,       #   | If stack == 1: [var6]
1, 3,           #   |   Push 3 [3, var6]
18,             #   |   Get string length 3 [len, var6]
4, 0, 5,        #   |   If len != 0: [len, var6]
1, var5,        #   |     Push '.' ['.', len, var6], {3: 'a'}
1, 3,           #   |     Push 3 [3, '.', len, var6], {3: 'a'}
12,             #   |     Append char to string [len, var6], {3: 'a.'}
0,              #   |   Drop stack [var6]
0,              #   | Drop stack []
                #
                #   Append string [3] to end of stack
                #
1, 3,           #   Push 3 [3] {3: 'def'}
13,             #   Append string [3] to end of stack ['d', 'e', 'f']
                #
                #   | Newly added routine: if stack not empty, switch to mode = 3
28,             #   | Get stack length [len]
4, 0, 5,        #   | If not len == 0:
                #   |   Change mode to 3
1, 3,           #   |   Push 3 [3, len]
1, 0,           #   |   Push 0 [0, 3, len]
8,              #   |   Put into dictionary [len], {0: 3}
0,              #   | Drop stack []
                #
5,              #   Break
                #
                # Else: Check if mode is already set - or set 0
1, 0,           #   Push 0 [0, c]
10,             #   Is in dictionary 0 [result, c]
3, 0, 5,        #   If result == 0: [0, c]
                #     ((we could drop 0 and add it here again...))
1, 0,           #     Push 0 [0, 0, c]
8,              #     Put into dictionary [c], {0: 0}
1, 0,           #     Push 0 [0, c]
0,              #   Drop stack [c]  <-- [CONTINUE] Jump here from above
                #
                #   Perform action based on mode: Get mode
1, 0,           #   Push 0 [0, c]
9,              #   Get from dictionary [mode, c], {0: mode}
                #
3, 0, 7,        #   If mode == 0: [0, c]  <-- read input mode
0,              #     Drop stack [c]
1, 1,           #     Push 1 [1, c], {1: 'a'}
12,             #     Append char to string [], {1: 'ac'}
1, var1,        #     Push var1 (reply separator) [' ']
5,              #     Break
                #
                #   | Newly added routine: Mode 3: fill waiting time with space
```

```
                    #   |
        3, 3, 10,   #   | Elif mode == 3: [3, c]  <-- wait for feedback mode
        0,          #   |   Drop stack [c]
        3, var4, 4, #   |   If c == ' ': [' ']  <-- teacher's reply separator
        0,          #   |     Drop stack []
        1, var1,    #   |     Push var1 (reply separator) [' ']  <-- student's reply
        5,          #   |     Break
                    #   |   Else:
        19, 9,      #   |     Goto Else below (Change mode to 2) (goto 140)
                    #
        3, 1, 13,   #   Elif mode == 1: [1, c]  <-- wait for feedback mode
        0,          #     Drop stack [c]
        3, var4, 2, #     If c == ' ': [' ']  <-- teacher's reply separator
        0,          #       Drop stack []
        5,          #       Break             <-- student stays quiet
                    #     Else:
                    #       Change mode to 2
        1, 2,       #       Push 2 [2, c]
        1, 0,       #       Push 0 [0, 2, c]
        8,          #       Put into dictionary [c], {0: 2}
        1, 0,       #       Push 0 [0, c]
                    #       Continue without return
                    #
                    # Operation in mode 2:
        0,          #     Drop stack [c]
        1, 2,       #     Push 2 [2, c], {2: 'a'}
        12,         #     Append char to string [], {2: 'ac'}
        1, var1     #     Push var1 (reply separator) [' ']
]

# The following annotations indicate, at what locations of the code the genetic code
# can be replaced by other code fragments with certain functions:
annotation = [
    [4, 5, Breeder.annotation_type_store_input_feedback_pair, 1, 2, 10],
    [29, 14, Breeder.annotation_type_store_input_separator_in_reply, 1, 2],
    [47, 1, Breeder.annotation_type_store_input_separator, 1, 2],
    [48, 0, Breeder.annotation_type_start_reply, 1, 2, 10],
    [58, 3, Breeder.annotation_type_lookup_reply, 1, 2, 3]
    [61, 0, Breeder.annotation_type_string, 3],
    [61, 18, Breeder.annotation_type_reply_separator, 3, var5]
]
```