# Machine Learning of Human Language through Automatic Programming

**Roland Olsson (Roland.Olsson@hiof.no)**
Department of Computer Science; Østfold College
Os Alle' 11, N-1757 Halden, Norway

**David Powers (powers@cs.flinders.edu.au)**
School of Informatics and Engineering; Flinders University
GPO box 2100, Adelaide, SA 5001, Australia

**Abstract**

Automatic programming, that is, machine synthesis of algorithms, has advanced to the stage where most simple standard algorithms, for example for searching, sorting and combinatorics, can be routinely synthesized. In this paper, we study a much more difficult problem, namely automatic programming for learning the semantics of human language. As far as we know, the most advanced and capable system for fully automatic programming is Automatic Design of Algorithms through Evolution (ADATE), which has a unique ability to generate recursive functional programs from first principles with automatic invention of recursive help functions. The semantics of the simple human language learnt by ADATE in our experiments is grounded in a desktop world where an agent moves a cursor on a surface covered with a number of windows, similar to the desktop facing millions of computer users every day.

## Introduction

When designing machine learning methods, it is desirable to have progressions of more and more difficult test problems that can be extended to the difficulty and variability of problems encountered by ourselves. Such progressions of machine learning problems can be obtained using a simulation of agents that communicate in increasingly larger subsets of human language. We discuss how to define problem progressions and present experimental results for the first few problems in a progression concerning manipulation of a Windows desktop.

In order to make computers learn a primitive human language and manipulate a desktop, we use Automatic Design of Algorithms through Evolution (ADATE) (Olsson, 1995) that generates purely functional programs.

We provide evidence supporting the hypothesis that comparatively simple evolution with great, but still feasible, time complexity can produce complex algorithms reflecting language and simulated worlds of increasing complexity.

Much of computer science is devoted to reducing the ratio between the complexity of computer languages and their expressiveness. In particular, purely functional programming languages, for example Haskell (Peyton-Jones, 2003) and Clean (Plasmeijer & Eekelen, 1993) were designed to reduce the complexity of programs and are ideally suited for complex symbolic processing of syntax and semantics for human language.

It is no coincidence that purely functional languages also are ideal for automatic evolution of algorithms, for example by facilitating program transformations. In this paper, we employ ADATE to evolve purely functional programs that represent the semantics of human language.

ADATE has previously generated a range of relatively small recursive programs in problem domains such as sorting, searching and combinatorics. It is a general system for automatic functional programming with the ability to evolve recursive programs from first principles and automatically invent recursive help functions. The main drawback, shared with all methods for program evolution, is a huge computational demand.

The primary restriction of our approach is this computational complexity rather than the ability to define progressions of problems, even if this is non-trivial and a research topic on its own. We provide experimental results as well as a theoretical discussion for the time complexity of ADATE.

## Progressions of learning problems

The problems and challenges posed by nature are open-ended in the respect that no known life form has perfect or nearly optimal solutions to all of them.

We seek a progression of well-defined machine learning problems that similarly is open-ended in the respect that no machine learning method can be expected to solve all of them. Being open-ended also means that if a machine learning method succeeds in solving all problems in the currently defined prefix of the progression, it should be straightforward to extend the prefix until the method fails.

The progression should also be smooth which means that minute fitness differences between individuals can be identified. Another desirable property is that the start of the progression consists of quite simple problems that can be solved by individuals with almost no cognitive ability at all.

Yet another desired property is that the problems should be well defined, facilitating analysis of their machine learnt solutions. For example, the problems in nature as well as in artificial life simulations are implicitly defined by the world and not well defined.

Thus a problem progression should have four properties to be suitable for driving artificial evolution namely *simple start*, *open end*, *smoothness* and *analyzability*. Natural language learning problems seem ideal with respect to these four criteria.

There are several dimensions along which to study the evolution of individuals with linguistic ability, for instance the following two.

1. Evolution of a language faculty for a fixed and genetically determined language versus evolution of a language acquisition device (LAD) (Kirby, 2002).

2. Evolution to learn a specific human language, for example English, versus evolution studying emergence of language (Steels, 1997).

We argue that evolution of functional programs is suitable for all points in this two-dimensional space. However, this paper focuses on direct evolution of a simple human language for the following two reasons.

1. Evolution of a LAD may be more computationally demanding and irregular, making it less useful for easily studying the complexity and scalability of evolution. Note that evolution of a learning algorithm, e.g. one similar to any of the main algorithms in (Mitchell, 1997), appears to be a minimum requirement.

2. It is easier to analyze the results of language learning without having to also deal with an emergent, probably unknown, alien and incomprehensible language. However, most experiments reported in the literature are so simple that this alien language problem is surmountable.

The analyzability alluded to above is yet another reason for using functional programs, particularly ones that are not too far from minimum size, instead of neural networks, say, which are more like black boxes.

## The desktop learning scenario

The semantics of the simple language in our experiments is grounded in a desktop world where an agent moves a cursor on a surface covered with a number of windows, similar to the desktop facing millions of computer users every day. The agent is supposed to respond, either by actions or words, to sentences from a simple compositional language containing a few verbs, prepositions and nouns as well as the words "yes" and "no"

For example, assuming that A and B are windows, the sentence "go inside A" should induce the agent to move the cursor so that it is inside A whereas the sentence "is above B" yields a "yes" or "no" reply and no action. This model is both simplistic and simple, in that it does not require the evolution of cooperative behaviour (Steels, 1997) but employs a direct fitness function.

The role of ADATE is to synthesize a function `f` that controls the agent by analyzing a sentence and producing either a next action or a reply. The main loop of an agent, written in the environment passing style of Clean, employs `f` together with a simple desktop simulator function, `nextWorld`, as shown in Figure 1. This function applies an action to a world and returns the resulting new world.

The parameter `Words` is a sentence represented as a list of words where each word consists of four bits. Thus, the smallest syntactic unit is a bit and not a word. `World` is a three tuple, `( Cursor, Windows, Count )`, where `Cursor` and `Windows` specify the current state of the desktop and `Count` is the number of actions that have been applied so far.

The value of `Action` returned by the synthesized function `f` is `right`, `up`, `left` or `down` and causes the corresponding movement of the cursor when fed to `nextWorld`. In future experiments, we plan to add actions `grab` and `release` enabling the agent to also move windows.

As seen in Figure 1, `f` may just return a list of words `Ws` which causes termination of `main` and provides `Ws` and the current state of `World` as evidence of the agent's work.

In this paper, we experimentally study a short progression of language learning problems made successively more complex by including more words chosen from the following small vocabulary.

**Prepositions.** `inside, outside, above, below, leftOf, rightOf`

**Nouns.** `A, B, C, D, E, F`

**Verbs.** `go, is`

**Interjections.** `yes, no`

The nouns are all windows and could of course have been named "Internet Explorer", "My Documents", "Acrobat Reader" and so on to increase the resemblance to a well known desktop that every reader of this paper is likely to know about.

## Experimental methods and results for the desktop world

An ADATE specification of a learning problem contains algebraic data types, training inputs, an evaluation function and a limit on the execution time of a synthesized program. We will first discuss the evaluation function and then our choice of training inputs but omit formal definitions of the types.

```
fun main( Words,  World ) =
  case f( Words, World ) of
    finished Ws => result( Ws, World )
  | some Action => main( Words, nextWorld( Action, World ) )
```

Figure 1: The main loop of an agent.

## The evaluation function

Since there typically are many action sequences that achieve a given goal, it is in general inappropriate to use input-output pairs as training data for an agent manipulating a world. However, the desktop learning problem described above is so simple that it would be possible to solve it with input-output pairs. This would greatly simplify the learning but also jeopardize the *open end* property of the problem progression since we expect that it would be more difficult to provide suitable explicit outputs instead of an evaluation function for more complex agent learning problems.

With our evaluation function the learner obtains feedback only after having completed an entire action sequence and replied in words. This is a form of so-called reinforcement learning which in general is both more widely applicable and more difficult than learning from explicit input-output pairs.

The evaluation function uses a predicate to check that a desired world state is obtained after simulating the agent for a maximum of 2000 time steps. Additionally, the evaluation function considers the number of moves made by the agent and the Hamming distance between the reply in `Ws` and a desired reply. Of course, both the number of moves and the Hamming distance are to be minimized.

One may ask if this Hamming distance resembles the feedback received by children when they learn language. Whilst "Poverty of the Stimulus" claims that children do not receive enough correction to explain learning and do not change their language model as a result of what they do get, this misses a number of important points. Children get implicit feedback as to whether their sentence was successful or not, whether they got what they asked for or something else, whether they were understood or misunderstood. Moreover there is a well known strategy of sentence repair whereby a parent/teacher reflects back what they understood in the correct form. Furthermore there is evidence of anticipated correction where the learner has a model based on sentences heard that is sufficient to realize that an utterance doesn't sound right and provides constraints that direct learning. For further discussion on this point see Powers and Turk (1989).

If the verb in a training input is `go`, the agent is not allowed to speak which means that the desired reply is the empty list `nil`. The verb `is` gives a desired reply that is a list containing the binary representation of either "yes" or "no".

In light of the discussion above regarding the lacking generality of explicit outputs, it is not entirely satisfying to explicitly specify these three reply alternatives. However, the final world state is certainly not explicitly specified. This shows up in the experiments where, for example, some of the intermediary programs always move the cursor to a corner of A in response to "go inside A". The best programs, on the other hand, choose the shortest path to the inside.

## The training inputs

We employ randomly generated combinations of initial worlds and sentences as training inputs. The total evaluation i.e., fitness, for a program synthesized by ADATE is computed by running the evaluation function discussed above for all training inputs and adding the resulting values.

We will now describe the probability distribution that is employed to generate training inputs. First of all, we choose a three word sentence of the form

```
<Verb> <Preposition> <Noun>
```

Each word is chosen uniformly at random from the alternatives listed in the vocabulary above.

The next step is to choose the windows. The lower left corner of a window is chosen uniformly at random on a desktop with a size of 100 times 100 pixels. The maximum width and height of the window is chosen uniformly on 1, 2, 3, 4, 5. The width and the height are then chosen uniformly on the interval from one to this maximum.

The coordinates for the one pixel cursor are chosen relative to the window denoted by the noun. With a probability of 1/3 the coordinates are chosen uniformly among the pixels in the window. With a probability of 2/3 the coordinates are chosen uniformly on a three pixel wide zone immediately outside the window.

The ADATE specification file discussed above as well as the source code for ADATE itself are available from Roland Olsson upon request.

## Resulting synthesized programs

Given specifications as described above, ADATE synthesizes compact and correct functional programs with a length of about 100 lines.

Initially, ADATE's population only contains the program shown in Figure 2. This program always

```
fun f( Words, World as world( Cursor as pixel( X, Y ), Windows, Count ) ) =
  raise D1
```

Figure 2: The initial program.

raises the exception `D1` which indicates that it does not know what output to produce.

After transforming this initial program a few hundred times under the guidance of the evaluation function described above, ADATE produces programs like the one partially shown in Figure 3, where the two occurrences of ... together represent about 80 lines of code that have been omitted to save space. This program is written in a small and purely functional subset of Standard ML.

The parameter `Windows` is an array of windows accessed by a subscripting operation not shown in Figure 3. This array cannot be updated by a synthesized program. The constructor `world` is not available to a program, implying that a new world can only be constructed by emitting an action. Lists of words are constructed using `nil` and `cons` that occur in the pattern matching case-expressions in Figure 3. Each element in such a list is a four bit word that is analyzed by the program using case-expressions like `case Vc48b3d of`.

It is interesting to see that the program in Figure 3 is recursive even if the function `f` can be defined without recursion. The recursion is a way for the program to talk to itself and ask questions like "How would I respond if someone said the following?". It could be possible for programs to develop their own internal language for recursive feedback, which would hinder analysis of the programs.

## Time complexity analysis and experiments

Long run time is the greatest weakness of ADATE, but it is still far ahead of exhaustive search and many other evolutionary algorithms. In this section, we will first describe the factors contributing to overall run time and then check how this analysis agrees with simple experiments somewhat analogous to desktop language learning. To analyze the time complexity, we need to consider ADATE's population structure and overall search. Keep in mind that neither can be exactly described in the small space available here.

ADATE's population consists of a back-bone of base programs in a size-fitness ordering such that a program in the back-bone always is better than all seen smaller programs. Thus a back-bone of programs $P_1, P_2, \ldots, P_n$ is such that each $P_i$ is a little better and a little bigger than $P_{i-1}$.

Roughly speaking, each base program corresponds to a species in nature. We will use the term neighbourhood instead of species to indicate the set of all individuals produced from and related to a base, most of which have almost the same fitness as the base.

For the simple analysis provided here, we assume that the run time of ADATE is proportional to the product of the number of training inputs, the max run time per input, the max neighbourhood cardinality and the total number of neighbourhoods.

In ADATE, the number of fitness values that occur is roughly equal to the number of base programs. After a fitness value has first occurred, it can only move downwards along the back-bone. Since the minimum size difference is about unity, say, the number of moves is limited by the maximum size of the first program with the fitness value. In a typical ADATE run, the maximum size of any produced program is of the same order as the number of bases. Thus it is reasonable to assume that the number of neighbourhoods that occur for a given fitness value is bounded by the number of bases and that the total number of neighbourhoods is proportional to the square of max program size.

The critical factor in this analysis is max neighbourhood cardinality. Let a site be a node in a program's syntax tree. ADATE will try program transformations at each site and iteratively deepen their complexity. For example, when synthesizing a permutation generation program, ADATE needs about one million transformations for a site in the worst case whereas only a few thousand are needed for the program in Figure 3 even though this program is a magnitude of order bigger.

Our conjecture is that the transformation complexity per site is bounded by a constant no matter how big and complex programs that are produced provided that the specification is written to facilitate evolution.

In the worst case, a program may be improved at only one site whereas a more favourable case is that the number of sites that can be immediately improved is proportional to program size. Thus we assume that max neighbourhood cardinality is proportional to program size in the worst case.

The conclusion is that overall run time in the worst case should be proportional to the product of the number of training inputs, the max run time per input and the cube of max program size.

Let us now check this experimentally. To obtain sufficiently many run time measurements with reasonable consumption of CPU cycles, we employed a highly simplified version of the desktop language learning problem where the objective is to learn a random mapping from binary words to constants.

```
fun f( Words, World as world( Cursor as pixel( X, Y ), Windows, Count ) ) =
  case Words of
    nil => (raise NA_474faf8)
  | cons( V172e44 as word( V172e45, V172e46, V172e47, V172e48 ), V172e49 ) =>
  case V172e49 of
    nil => finished( Words )
  | cons( Vc48b3c as word( Vc48b3d, Vc48b3e, Vc48b3f, Vc48b40 ), Vc48b41 ) =>
  ...
          f( case ( V6206a7f < X ) of
              false => (
                case Vc48b3d of
                  0 => V63436c2
                | 1 => ...,
             World )
```

Figure 3: Fragments of a correct program.

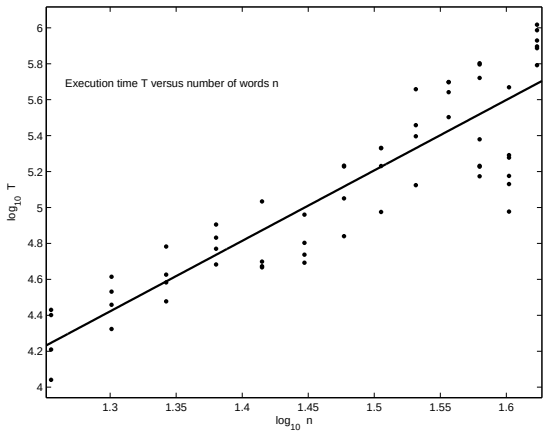


Figure 4: Least squares fit for execution time.

Note that this is a sub-problem of the desktop problem above corresponding to learning the mapping from the binary representations of nouns to indices in the array of windows.

We chose a word length of eight bits and fifty constants for this decoding problem and ran it a number of times for each number of words $n$ in $18, 20, 22, \ldots, 42$. The training data for each run was simply the $n$ input-output pairs that a random mapping consists of. Even if it is trivially easy to write a specialized machine learning algorithm for this decoding problem, it is still meaningful to use it for basic testing of the time complexity of a general machine learning system like ADATE.

The programs synthesized for $n \geq 40$ have about the same size as the program in Figure 3 but a rather trivial logical structure. The experiments were carried out on four 1.1 GHz Athlon PCs and took a total run time of a few months. The base 10 logarithm of the run times for the decode experiments are plotted against the base 10 logarithm of $n$ in Figure 4. Assume that the run time $T(n) = cn^k$ for some constants $c$ and $k$. Since $\log T(n) = k \log n + \log c$, the gradient of the straight line in the figure is an approximation of $k$.

This straight line was fitted to the data using least squares with bisquare weights in Matlab's curve fitting toolbox. Least squares assumes a normal distribution with the same variance for each $n$. Obviously, this is not true for most run time estimation such as the one attempted here since the variance increases with $n$. The bisquare weights are a primitive attempt to compensate for the non-constant variance. Since we obtained quite similar fits also with other least square methods, the straight line in Figure 4 is unlikely to grossly misrepresent the real expected run time as a function of $n$.

Matlab reports that $k$ is 3.9 with 95% confidence bounds of 3.3 and 4.5. Above, we said that worst case run time should be proportional to the product of the number of training inputs, the max run time per input and the cube of max program size. In the decode experiments, the program size is proportional to $n$, that is, the number of training inputs. Since the max run time per input is constant, this gives a worst case run time of $o(n^4)$ whereas our measurement above is $o(n^{3.9})$. It is not yet clear why the measured time is so close to the worst case, but the agreement between theory and this simple experiment is nevertheless good.

An interesting question is how the run time for the desktop learning problem depends on the number of nouns and the number of prepositions. The desktop experiments were run with 800 training inputs on a cluster of sixteen 800 MHz Pentium III machines. Run times varied between one and several weeks on this cluster when the number of prepositions plus the number of nouns varied between four and twelve. Since the number of inputs were about twenty times

as many as for the biggest decode runs, we would expect run time to be about twenty times longer given that max program size and max run time per input are about the same

The biggest decode runs consumed a single-CPU time of slightly more than one week. Once again, agreement between theory and experiments is reasonable since this would lead us to expect about two weeks on the cluster for the desktop runs.

## Conclusions

We have examined the use of a functional formalism for representing, learning and evolving language using the ADATE automatic programming system, and have benchmarked and analyzed its performance in two sets of experiments. One set aimed to assess its applicability to language learning and learned semantic relationships in the context of a single syntactic frame and an elementary simulated desktop environment; the second aimed to confirm our analysis of the efficiency of the approach and involved learning a random association of symbols and meanings. Whereas exhaustive blind search would have to sift through $n^n$ or $n!$ possible combinations of associations (with and without replacement resp.), both our theoretical and empirical analyses demonstrate a worst case $o(n^4)$ performance for ADATE on this task with its $n$ inputs and $o(n)$ expected program size, and an effective limitation of the number of "neighbourhoods" or "species" considered to $o(n^2)$.

In the desktop semantics experiments, program size and runtime/input were held roughly constant, demonstrating pleasing efficiency with theoretical and empirically verified runtimes of $o(n)$ for $n$ inputs. The efficacy of the system was exemplary, synthesizing correct programs of around 100 lines for each example. These programs are quite compact, however the formalism does not lend itself to ease of readability. Interestingly, programs tended to be recursive even though recursion was not required for a correct solution. The recursion seems to have a meta-level role in allowing the program to consider potential actions or responses before making a final judgement. This behaviour is reminiscent of theories of anticipated correction in which a language learner contemplates or even makes an utterance, but then realizes that it doesn't "sound right" and repairs the sentence. See Powers and Turk (1989) for a detailed discussion of explicit and anticipated correction as an alternative to Chomskian nativist doctrine that language is evolved rather than learned. In short, there is no need to assume an innate language acquisition device uniquely specialized for language but capable of learning arbitrary human languages; nor is there a need to assume a language-like interlingua as a common representation across speakers of different languages.

In formulating these experiments the decision was made not to seek to evolve a language acquisition device (LAD) but rather the aim was to evolve a language directly. For this reason the performance of this evolutionary approach is reminiscent of machine learning approaches more than evolutionary modeling, though we note that evolutionary learning is a special case of machine learning and differs primarily in the nature and use of the evaluation or fitness function. The difference between learning and evolution is apparent in that although the evaluation function assumes quantifiable feedback it does not explicitly deny "poverty of the stimulus" and this is reflected in the fitness of the individual rather than in explicit correction. There is in particular no explicit pressure to evolve a program that incorporates a correction model, an interlingua or a language acquisition device, which makes it especially interesting that a recursive model is developed that hints at the emergence of an interlingua or LAD.

## References


Kirby, S. (2002). Natural language from artificial life. *Artificial Life, 8(2)*, 185–215.

Mitchell, T. (1997). *Machine learning.* McGraw Hill.

Olsson, R. (1995). Inductive functional programming using incremental program transformation. *Artificial intelligence, 74(1)*, 55–83.

Peyton Jones, S. (2003). *Haskell 98 Language and Libraries* Cambridge University Press,

Plasmeijer, R., & van Eekelen, M. (1993). *Functional Programming and Parallel Graph Rewriting.* Addison Wesley.

Powers, D,. & Turk, C. (1989). *Machine learning of natural language.* Springer-Verlag.

Steels, L. (1997). The synthetic modeling of language origins. *Evolution of Communication, 1*, 1–34.