Transfer Learning via Modularity in a Model of Technological Evolution

Eric Scott\textsuperscript{a, b}

\textsuperscript{a}Department of Computer Science, Andrews University, 4260 Administration Drive, Berrien Springs, MI 49104
\textsuperscript{b}Santa Fe Institute, 1399 Hyde Park Road, Santa Fe, NM 87501

Abstract

Most machine learning algorithms ultimately focus on optimizing solutions to a single target function. Cooperative phenomena that transfer information across distinct environmental niches, however, lie at the heart of the evolution of complex functions in nature and technology, where solutions adapted for one problem are repurposed to solve another, related problem. Boolean functions have become a popular toy model for exploring the dynamics of such processes, and provide insight into new approaches to evolutionary computation. We implemented the a model of combinatorially evolving logic circuits developed by Brian Arthur and Wolfgang Polak in which solutions to boolean functions are encapsulated as modules that can be used to solve other, more complex functions, and began to explore the sort of transfer phenomena its success depends upon. We observed a significant difference in the dynamics of evolution between when the full suite of fitness functions is present, versus when only a few pieces of the selective pressure are active at a time. Future work is needed examine these dynamics in more detail.

April 1, 2011
1. Introduction

In both natural and human problem solving, local optima are overcome by transfer: using knowledge gained in one domain to facilitate the learning of a task in a second, distinct target domain. Biological evolution utilizes Darwinian preadaptation (a.k.a. exaptation) to build sophisticated mechanisms by coopting existing artifacts\[17, 21\], and builds a repertoire of biomolecular artifacts that are copied and reused extensively for myriad functions within the same organism\[4, 9, 27\]. In this article we will use exaptation and transfer interchangeably. The computational model we study is motivated by the evolution of technology in the human economy, which can be seen to advance via a similar process: inventions are frequently combined with other technologies and repurposed for a use other than the original motivation\[2\].

In these views of problem solving, the evolutionary history of a solution plays a significant role in determining what form it takes and whether the solution is likely to successfully evolve at all. A diversity of niches makes some otherwise difficult problems easier to solve (positive transfer) while making others more difficult (negative transfer). Evolution is here seen as a process which “tinkers” with new combinations of old material more often than generating new information \textit{ex nihilo} \[19, 38\]. The potential gains in problem solving efficiency offered by this paradigm motivate an approach to artificial intelligence which seeks to exploit historical contingency.

\textit{Transfer learning} (TL) has recently become an active area of machine learning research\[28, 35, 36, 32\], aiming at the reuse of information across problem domains. Central questions in TL include how to know when transfer can be beneficial for solving a target task, what knowledge to transfer, and how to ensure that negative transfer does not occur. Transfer learning is distinct from dynamic environments\[8\] (fitness functions that change over time), coevolutionary algorithms\[12\] (in which multiple solutions compete or cooperate interactively), multiobjective optimization (where multiple conflicting objectives are pursued simultaneously)\[37\], and mixtures thereof (for example, \[16\]), in that none of these methods have the goal of utilizing exaptive analogies between distinct problems and, with the exception of coevolution, neither do they attempt to implement intermediate developmental steps as the dominant solution mechanism.

The importance of diverse fitness environments to evolution of complex and/or robust features has been observed in many computational models\[5, 6, 7, 13, 18, 23, 24, 26, 29, 31, 33, 34\]. Logic circuits and other genetic programming paradigms\[22\] have often provided a sandbox for exploring the dynamics of evolution in part because of the ease with which they can be analyzed, but also because of the (limited) analogy they provide to genetic regulatory networks\[20\]. With these Macia and Solé\[24\] have drawn attention to the role degeneracy plays in organism robustness, while Parter, Kashtan and Alon\[29\] have modeled the theory of Facilitated Variation\[15\] to show the spontaneous development of modularity under exposure to multiple, distinct fitness environments. Similarly, Lenski et al.\[23\] have used the \textit{Avida} artificial life simulator to highlight the importance of intermediate logic functions in the evolution of solutions to more complex functions.

An open problem in computational research on transfer and exaptation is whether similarity metrics can be devised which predict in advance whether positive or negative transfer can be expected and to what degree. Some metrics have been proposed for approaching this problem (\[11, 25\]), based for instance on Kolmogorov Complexity, but more empirical work on transfer is needed before effective metrics can be discovered, and before the potential and limitations of transfer can be understood.

We analyze the evolution of boolean functions in a model of technological evolution developed by Arthur and Polak\[3\], with the ultimate goal of identifying the potential of one logic function being coopted for the development of another. Based on an inherently modular design, this model underscores the role modularity can play in transfer across domains with different dimensionalities. We chart sequences through this path-dependent space of selection pressures in analogy to the transformations undergone by reactants in a chemical network, so that we can construct a Petri net to graphically represent the relationships
between environments. Here, fitness functions act as enzymes, with solutions acting as products and reactants.

1.1. ANN Illustration

First, we illustrate transfer and show with a toy example, and show how positive and negative transfer relationships can be mapped between a set of fitness functions. A simple ANN was trained via backpropagation to simulate the OR, XOR, and COUNTONES (count the number of ones in the input signal) functions for a two-bit input string (See Figure 1).[30]

The network is initialized to state (“species”) $E_0$ with random weight values on $(-0.1, 0.1)$. We may then train it on the XOR function to create the “species” $X_0$. The $X_i$’s represent solutions that solve the XOR problem fairly well, $O_i$’s solve OR, $C_i$’s solve COUNTONES, and $D_i$’s fail to solve any of the three. Now, if we use $X_0$ as the initial condition to train a solution $O_1$ to OR, it takes only one training cycle of backpropagation. Since this is significantly shorter than the number of cycles required to train $O_0$ from $E_0$, we say that $X_0$ can be exapted by OR, i.e. positive transfer occurs. Alternatively, one could say that XOR catalyzes the production of a solution to OR. Using $O_1$, then, as the initial condition to train a new solution to XOR, we get a cycle that may be repeated ad infinitum (See Figure 2a). On the other hand, if we train $E_0$ on OR first, and use the resulting solution $O_0$ as the initial condition to train XOR, we see that the algorithm never converges onto a solution to XOR – negative transfer occurs.

Exaptive relationships between the three objective functions are represented by the complicated graph.
in Figure 3, imitating the visualization of chemical networks.[14] The green edges point from the fitness function to the species that was optimized on it. The edge labels denote the initial condition used to create the species. Note that solutions to the same problem generated from different initial conditions (ex. \{O_0, O_1, O_2\}) display different exaptive (chemical) properties. Two or more solutions are represented by the same node in the graph if they display similar chemical properties.

Figure 3:

2. Model

The model developed by Arthur and Polak[3] is unique in that the only evolutionary operator is the combination of existing circuits (i.e. the composition of boolean functions). Since its primary motivation is to explore the path-dependent dynamics of technologies, as engineers synthesize tools and components that were developed in different environments to meet separate needs, there is no mutation operator. This divergence from the biological paradigm comes with a benefit in that, by composing boolean functions of different dimensionality, solutions can pass through fitness functions of different dimensions, thus broadening the class of functions amongst which we can look for exaptive effects. For example, if the function \(A : \mathbb{B}^m \mapsto \mathbb{B}^n\) has been evolved to solve fitness function \(\alpha\) that maps \(m\) inputs to \(n\) outputs, it can be composed with some other function into a new circuit \(B : \mathbb{B}^p \mapsto \mathbb{B}^q\) and tested against a function \(\beta\) with a different number of inputs and outputs.

The algorithm proceeds as follows:

- Initialize a set primitives to contain only NAND.
- Initialize a pool of circuits to be empty.
- Select from 2–12 components from the primitives set and/or pool with a weighted choice function.
- Randomly wire together \(n\) variations of the circuit
- For each variation:
  - Evaluate the average fitness against a battery of fitness functions.
  - If the variation’s truth table completely matches one of the goals, add it to the primitives set.

Components were selected randomly from the primitives set with a probability of 0.8. Otherwise, a component was selected from the pool via tournament selection on average fitness.

We aim to answer questions like:

- Does the presence of an AND circuit in the primitives set facilitate the evolution of a HALF-ADDER circuit?
- Does the presence of both AND and HALF-ADDER fitness functions as a multiobjective selection pressure facilitate the evolution of a HALF-ADDER?
HALF-ADD
ONLY NAND HALF-ADDER
WITH_AND_C NAND, AND HALF-ADDER
WITH_OR_C NAND, OR HALF-ADDER
WITH_OR_CN NAND, OR HALF-ADDER

3. Results

Several scenarios were constructed with different initial sets of primitives and suites of fitness functions. The rate at which desired functions were evolved was then measured over several thousand generations.

In all the scenarios tested besides FULL, the mean occurrence rate for 5,000 generations was close to 150 with a standard deviation close to 10. No statistically significant differences between the scenarios were detected. A broader family of scenarios will need to be analyzed in the future to determine the sort of interactions that lead to the reduction of occurrences in the FULL scenario.

In general, our simulation does not converge on solutions as fast as Arthur and Polak’s. This may have to do with differences in our selection mechanism, or with parameters, or it could be a bug in our code.

4. Discussion

We implemented the basics of Arthur and Polak’s model, and began to explore the sort of exaptive phenomena its success depends upon. We observed a significant difference in the dynamics of evolution between when the full suite of fitness functions is present, versus when only a few pieces of the selective pressure are active at a time. We determined that there is need for further analysis of more complex scenarios before we can map the sort of relationships we observed in the Artificial Neural Network example. A method to follow can be found in [26], where a game-theoretic Shapely analysis was used to evaluate the contribution of different evolutionary operators acting on neural networks learning via exaptation. This could be used to evaluate the contribution of each available module to the evolutionary process, though it may prove computationally intractable.

In the future we would aim to map these dependencies in detail, and to answer the question:

- Are the positive/negative transfer rates of fitness functions and/or modules greater than the sum of their parts in this model?

5. Implementation

The simulation was implemented in Common Lisp, and the implementation technicalities discussed below. The full code is available at [http://github.com/SigmaX/CircuitTech](http://github.com/SigmaX/CircuitTech).

5.1. Representation

So that the substructure of a circuit can be easily retrieved for visualization, we represent it as an adjacency matrix. Since the components of a circuit may have multiple inputs and outputs, a digraph of functional dependencies between components is insufficient, since it would not specify which output of a component fed into which input of the next. We thus map outputs onto inputs directly in our matrix. For example, the circuit shown in Figure 5 is represented as follows:

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Primitives</th>
<th>Fitness Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALF-ADD</td>
<td>NAND</td>
<td>HALF-ADDER</td>
</tr>
<tr>
<td>WITH_AND</td>
<td>NAND, AND</td>
<td>HALF-ADDER</td>
</tr>
<tr>
<td>WITH_OR_C</td>
<td>NAND, OR</td>
<td>HALF-ADDER</td>
</tr>
<tr>
<td>WITH_OR_CN</td>
<td>NAND, OR</td>
<td>HALF-ADDER, OR</td>
</tr>
</tbody>
</table>

where \( \{x, y\} \) are the inputs to the circuit and \( \{O_1, O_2\} \) are its outputs. The labels are omitted in memory: a corresponding component vector is defined from which they can be generated.

Note that each component input can receive a signal from only one source, and that the corresponding functional dependency graph (in which each node is a component and edges denote any dependency) will always be acyclic in the circuits we are generating. We accomplish this as follows: A random ordering of components is selected, and both the rows and columns follow this order. We begin with the zero matrix. For each column vector of the matrix corresponding to an input \( A_i \) of component \( A \), a random element between the first row and the first output
The circuit outputs \( O_i \) have any element set to one. This ensures that the corresponding functional dependency graph has an upper triangular matrix with a zero diagonal, i.e. that it is acyclic. Since the ordering of the components was selected randomly, the resulting circuit is also random.

Executing circuits is expensive, especially since each component may itself be a circuit with many subcomponents, and so on. It is thus to our benefit to store the truth table of a circuit, so that its output can be used more than once without executing the function repeatedly. We generate a graph representation of the circuit’s truth table known as a Binary Decision Diagram (BDD), which represents the if-then-else normal form of a boolean function in a binary tree structure that can be easily walked for all input bit string permutations\([1]\). The BDD is compressed into a memory efficient Reduced Ordered Binary Decision Diagram (ROBDD), much like a prefix tree for a dictionary with an n-ary (as opposed to binary) alphabet can be compressed into a Directed Acyclic Word Graph\([10]\). ROBDDs have the added quality that two logic functions that are isomorphic (perform the same computation on a set of input variables) have ROBDDs that are not only isomorphic but identical. The ROBDD for functions with more than around sixteen variables can still become quite large, as the number of possible bit string permutations is \( 2^n \). It is computationally intractable to generate the entire truth table of functions much larger than this.

ROBDDs can be composed efficiently, and Arthur and Polak use such operators to generate new circuits. Our system composed functions via the generation of adjacency matrices and Lisp functions as described below.

5.2. Execution Tiers

To generate the ROBDDs for circuits, we need code that executes the function. To achieve this we automatically generate a Lisp function from the circuit’s adjacency matrix.

```
(defun one_b (x y)
  (apply #'(lambda (u v)
    (list (funcall B u v) (funcall C u v)))
  (funcall D x y)))
```

Note that we used a lambda expression in circuit \( I_1 \) so that the outputs of \( D \) could be used multiple times. More complex functions may require several nested lambda expressions, each of which must be executed in sequence, its outputs providing the inputs for the next function. We conceptualize each lambda as an execution tier, and consider each tier \( \tau \) to be a set of component functions. Determining a circuit’s execution tiers is the first step toward generating its Lisp expression, and is equivalent to performing a topological sort on the components (though, again, the circuits resist a straightforward graph representation).

Formally, we may define a tier as follows. Let \( \mathbb{C} \) be the set of component functions used in the circuit, ex. \( \mathbb{C} = \{A,B,C,D\} \), each of which in turn is a set of input variables \( X_i \). Let \( \phi_{X_i} \) be the column vector of the adjacency matrix corresponding to \( X_i \), with \( \phi_{X_i}(z) \) being the matrix element corresponding to output \( z \) (ex. In Figure 5, \( \phi_{A_1}(x) = 1 \), but \( \phi_{A_1}(y) = 0 \).)

**Definition 1.** A component function \( X \) is initially in tier \( \tau_0^1 \) if and only if \( X \)'s outputs depend only on the circuit’s inputs \( x, i \in I_1 \). That is, all the entries in each \( X_i \)'s column vector are zero except those corresponding to inputs to the circuit, i.e.:

\[
\tau_0^1 \equiv \{ X \in \mathbb{C} | \forall X_j \in X, (\forall(z \in \phi_{X_j}) z \notin I_1, \neg \phi_{X_j}(z)) \}.
\]

Now, visualizing higher tiers as subcircuits with the outputs of the previous tier as inputs, if we define the outputs of the functions in \( \tau_0^1 \) unioned with \( I_1 \) as \( I_2 \) and so on, we can define the \( I \)th tier:

\[
I_{i+1} \equiv I_i \cup \{ Y_o | (\forall(Y_o) \in \tau_i^1) \land (\exists X_j \in X \not\in \tau_i^1, \phi_{X_j}(Y_o)) \}
\]

\[
\tau_0^i \equiv \{ X \in \mathbb{C} | \forall X_j \in X, (\forall(z \in \phi_{X_j}) z \notin I_i, \neg \phi_{X_j}(z)) \}.
\]

The subscript 0 signifies that this is not our final definition.
Equation (3) works for the circuit in Figure 5, but it is not general. A special case must be compensated for, as demonstrated by the circuit in Figure 8. Component B in Figure 8 is a part of the first tier, but depends on the output of A, contrary to our definition in (1). Since the output of A is used only once, we do not need a lambda function (tier) to process it. To complete the definition of \( \tau \) then, we define an iterative algorithm to add components akin to B. First we define three helper functions:

**Definition 2. Object and Usage functions:**

- A component \( X \) is in the set \( \text{Obj}(X_O) \) iff \( X_O \) is an output for \( X \).
- \( \text{Usage}(X_O) \) is the number of edges directed out of \( X_O \), i.e.
  \[
  \text{Usage}(X_O) \equiv \sum_i \phi_i(X_O)
  \]
- \( \text{ObUsage}(X) \) is the number of distinct components the outputs of component \( X \) are used in.

If a function’s output is used more than once, we need a new tier to store the value for processing multiple times. Furthermore, if a component’s outputs are directed into different child components, complex car and cdr arrangements can appear in the expression which are difficult to automatically generate. Thus we also require a new tier if a component’s \( \text{ObUsage} \) is greater than one:

**Definition 3.** \( Y \) is appended to \( \tau^i \) iff for all signals \( X_{Oj} \) directed into \( Y \):

- \( (\text{Obj}(X_{Oj}) \in \tau) \lor (X_{Oj} \in I_i) \)
- \( X_{Oj} \) is used only once, i.e. \( \text{Usage}(X_i) = 1 \).
- \( X \) has only one child components, i.e. \( \text{ObUsage}(X) = 1 \).

Combining these yields the following recurrence relation, to be executed until \( \tau^i \) reaches a steady state:

\[
\tau^i = \tau^i \cup \{Y \in \mathbb{C} | \forall Y_i \in Y_i, \forall (X_{Oj} | \phi_Y(X_{Oj})), (\text{Obj}(X_{Oj}) \in \tau^i) \lor (X_{Oj} \in I_i) \land [\text{Usage}(X_{Oj}) = 1] \land [\text{Usage}(|\text{Obj}(X_{Oj})) = 1] \}
\]

**References**


Figure 4: The results of one trial of independent evolutions of a HALF_ADDanalogousER in 5,000 generations under various scenarios. With all fitness functions active and only NAND in the initial primitives set, very few occurrences occurred. Roughly 150 occurrences appeared under a variety of simpler scenarios.
Figure 5:

Figure 6: Circuit 1a (left) and 1b (right).

Figure 7: The decomposition of a circuit into three execution tiers.

Figure 8: 