CROSS-TASK CODE REUSE IN GENETIC PROGRAMMING APPLIED TO VISUAL LEARNING

Wojciech Jaśkowski, Krzysztof Krawiec, Bartosz Wieloch

Institute of Computing Science, Poznan University of Technology, Poland
e-mail: `{wjaskowski,kkrawiec,bwieloch}@cs.put.poznan.pl`

We propose a method that enables effective code reuse between evolutionary runs that solve a set of related visual learning tasks. We start with introducing a visual learning approach that uses genetic programming individuals to recognize objects. The process of recognition is generative, i.e., requires the learner to restore the shape of the processed object. This method is extended with a *code reuse* mechanism by introducing a *crossbreeding* operator that allows importing the genetic material from other evolutionary runs. In the experimental part, we compare the performance of the extended approach to the basic method on a real-world task of handwritten character recognition, and conclude that code reuse leads to better results in terms of fitness and recognition accuracy. The detailed analysis of the crossbred genetic material shows also that code reuse is most profitable when the recognized objects exhibit visual similarity.

Keywords: Genetic programming, code reuse, knowledge sharing, visual learning, multi-task learning, optical character recognition

1. Introduction

Experimental evaluation of evolutionary computation methods, similarly to other computational intelligence approaches, usually concerns *isolated* tasks, often given in a form of well-defined, standardized benchmarks. Though justified by the scientific rigor of reproducibility, this happens to focus the research on the algorithms that match such experimental framework, i.e., on the methods whose 'lifecycle' is limited to a single experiment. Each experiment, typically composed of an evolutionary run on a particular problem and examination of its outcomes, takes place in isolation from the other experiments. The algorithm is not enabled to 'recall' its experience with other problems.

This approach is fundamentally different from the human way of problem solving, which is strongly based on experience. Priors in human reasoning come from one's history of dealing with similar tasks and are essential for acquiring new skills. By reusing knowledge, humans solve problems they have never faced before and perform well in difficult circumstances, e.g., in presence of noise, imprecision, and inconsistency.

The inability to make use of the past experience affects, among others, the genetics-based machine learning that this paper is devoted to. In a typical scenario, the learning algorithm (inducer) uses exclusively the provided training data to produce a *classifier*. In that process, the inducer relies on fixed priors that do not change from one learning process to another. There is no widely accepted methodology for reusing the knowledge that the inducer could have acquired in the past, and hence knowledge reuse is still listed among the most challenging issues in machine learning (Mitchell, 2006).

In an attempt to narrow the gap between the human and the machine way of learning, we propose here a simple yet effective approach to knowledge reuse. Our major contribution is a method of *code reuse* for genetic programming (Koza, 1992; Langdon and Poli, 2002; O'Neill, 2009), which operates between evolutionary runs that learn to recognize different visual patterns given by disjoint training sets. The technical means for that is a *crossbreeding operator* that allows individuals to cross over with the individuals evolved for other learning tasks. By way of learning task (or *task* for short) we mean the process of learning a particular class of visual patterns.

Within the proposed framework, such a task corresponds to a single evolutionary run that optimizes a specific fitness function.

The remaining part of this paper starts with a short review of related work and motivations for code reuse (Section 2). In Section 3 we detail the approach of generative visual learning based on genetic programming individuals that process visual primitives. Section 4 is the core part of this paper and presents the proposed method of code reuse within the genetic programming paradigm. Section 5 describes the computational experiment, and Section 7 draws conclusions and outlines the future research directions.

2. Motivations and Related Work

Koza (1994) made the key point that the reuse of code is a critical ingredient to scalable automatic programming. In the context of genetic programming, code reuse is often associated with knowledge encapsulation. The canonical result in this field are Automatically Defined Functions defined by Koza (1994, section 6.5.4). Since then, more research on encapsulation (Roberts *et al.*, 2001) and code reuse (Koza *et al.*, 1996) has been done within the genetic programming community. Proposed approaches include reuse of assemblies of parts within the same individual (Hornby and Pollack, 2002), identifying and re-using code fragments based on the frequency of occurrences in the population (Howard, 2003), or explicit expert-driven task decomposition using layered learning (Hsu *et al.*, 2004). Among other prominent approaches, Rosca and Ballard (1996) utilized the genetic code in form of evolved subroutines, Haynes (1997) integrated a distributed search of genetic programming-based systems with collective memory, and Galvan Lopez *et al.* (2004) reused code using a special encapsulation terminal. In a recent development, Li *et al.* (2012) proposed a code reuse mechanism for variable size genetic network programming.

In the research cited above, the code was reused only within a *single* evolutionary run. Surprisingly little work has been done in genetic programming to reuse the code between *multiple* tasks. To our knowledge, the first to notice this gap was Seront (1995), who investigated code reuse by initializing an evolutionary run with individuals from the concept library consisting of solutions taken from other, similar tasks. He also mentioned the possibility of introducing a special mutation operator that would replace some subtrees in population by subtrees taken from the concept library, in a way similar to our contribution, but did not formalize nor computationally verify it. An example of other approach to reusing the knowledge between different tasks is Kurashige's work on gait generation of a six-legged robot (Kurashige *et al.*, 2003), where the evolved motion control code is treated as a primitive node in other motion learning task.

In machine learning, the research on issues related to knowledge reuse, i.e., meta-learning, knowledge transfer, and lifelong learning, seem to attract more attention than in evolutionary computation (Vilalta and Drissi, 2002, for a survey). Among these, the closest machine learning counterpart of the approach presented in this paper is *multitask learning*, meant as simultaneous or sequential solving of a group of learning tasks. Following Caruana (1997) and Ghosn and Bengio (2000), we may name several potential advantages of multitask learning: improved generalization, reduced training time, intelligibility of the acquired knowledge, accelerated convergence of the learning process, and reduction of the number of examples required to learn the concept(s). The ability of multitask learning to fulfill some of these expectations was demonstrated, mostly experimentally, in different machine learning scenarios, most of which used artificial neural networks as the underlying learning paradigm (Pratt *et al.*, 1991; O'Sullivan and Thrun, 1995).

In the field of genetic algorithms (Holland, 1975), the work done by Louis and McDonnell (2004) resembles our contribution the most. In their Case Injected Genetic Algorithms (CIGAR) experience is stored in a form of solutions to problems solved earlier ('cases'). When confronted with a new problem, CIGAR evolves a new population of individuals and periodically enriches it with such remembered cases. The experiments demonstrated CIGAR's superiority to genetic algorithm in terms of search convergence. However, CIGAR injects *complete* solutions only and requires the 'donating' task to be finished before starting the 'receiving' task, which makes it significantly different from our approach.

From another perspective, our algorithm performs visual learning and that makes it related to computer vision and pattern recognition. Most visual learning methods proposed so far operate at a particular stage of image processing and analysis (like local feature extraction, e.g., (Perez and Olague, 2013; Chang *et al.*, 2010a)), which enables easy interfacing with the remaining components of the recognition system; using a machine learning classifier to reason from predefined image features is a typical example of such an approach. In contrast to that, we propose a learning method that spans the *entire* processing chain, from the input image to the final decision making, and produces a complete recognition system. Former research on such systems is rather limited (Teller and Veloso, 1997; Rizki *et al.*, 2002; Howard *et al.*, 2006; Tackett, 1993). Bhanu *et al.* (2005) and Krawiec and Bhanu (2005) proposed a methodology that evolved feature extraction procedures encoded either as tree-like or linear genetic programming individuals. The idea of using genetic programming to process attributed visual primitives, presented in the following section, was explored for the first time by Krawiec (2006) and further developed by Jaśkowski *et al.* (2007c) and Krawiec (2007).

Previously (Jaśkowski *et al.*, 2007a), we demonstrated the possibility of explicit cross-task sharing of code between individuals. Here, we study a method that benefits from code fragments *imported* from other individuals. In comparison with our earlier work on this method (Jaśkowski *et al.*, 2007b) here we additionally demonstrate its validity on a nontrivial, large multi-class problem, compare it to other machine learning techniques, analyze in-depth the effects of code reuse, provide extensive results for compound recognition systems, and place this study in the context of related research.

3. Generative Visual Learning

The approach of *generative visual learning* (Jaśkowski *et al.*, 2007c; Krawiec, 2007) evolves individuals (learners) that recognize objects by reproducing their shapes. This process is driven by a fitness function that applies a learner to each training image independently and verifies its ability to recognize image content. The process of recognition of a single training image starts with transforming it into a set of visual primitives (Alg. 1). Then, the learner is applied to the primitives and hierarchically groups them according to different criteria. At every stage of grouping process, the learner is allowed to issue drawing actions that are intended to reproduce the input image. Reproduction takes place on a virtual canvas spanned over the input image. The fitness function compares the contents of the canvas to the input image, and rewards individuals that provide high quality of reproduction. The trained learner may be then used for recognition, meant as discriminating the instances of the shape class it was trained on from the instances of all other classes (e.g., telling apart the examples of specific handwritten character from the examples of all the other characters).

The generative approach incites each individual to prove its 'understanding' of the analyzed image, i.e., its ability to (i) decompose the input shape into components by detecting the important image features, and (ii) reproducing the particular components. An ideal individual is expected to produce a

Algorithm 1 The process of recognizing image s by learner L

```
1: function RECOGNIZE($L$, $s$)
2:     ▷ $L$ - evaluated program (learner)
3:     ▷ $s$ - input image
4:     $P$ ←EXTRACTPRIMITIVES($s$)
5:     GROUPANDDRAW($L$,$P$,$c$)          ▷ $c$ - canvas
6:     return SIMILARITY($c$,$s$)
7: end function
```

minimal set of drawing actions that *completely* and *exactly* reproduce the shape of the object being recognized. These desired properties of drawing actions are promoted by an appropriately constructed fitness function. Breaking down the image interpretation process into drawing actions allows us to examine individual's processing in a more thorough way than in the non-generative approach, where the individuals are expected to output decision (class assignment) only. Here, an individual is rewarded not only for the final result of decision making, but for the entire 'track' of reasoning process. Thanks to that, the risk of overfitting, so grave in learning from high-dimensional image data, is significantly brought down.

Following Krawiec (2007), to reduce the amount of processed data, our learners receive only preselected salient features extracted from the original raster image s (function EXTRACTPRIMITIVES in Alg. 1). Each such feature, called *visual primitive* (or *primitive*) in the following, corresponds to an image location with a prominent 'edgeness'.

Note that our feature extraction method is simple and could be improved by applying more accurate edge detection algorithms as described by, e.g., Fabijanska (2012), or by using more sophisticated interest point detector, e.g., proposed by Trujillo and Olague (2006).

Each primitive is described by three scalars called hereafter *attributes*; these include two spatial coordinates of the edge fragment (x and y) and the local gradient orientation. The complete set P of primitives, derived from s by a simple procedure (Jaśkowski *et al.*, 2007c), is usually much more compact than the original image, yet it well preserves its sketch. The top part of Fig. 1a presents an exemplary input image (s in Alg. 1), and the lower part shows the set of primitives P retrieved from that image.

The actual process of recognition (procedure GROUPANDDRAW in Alg. 1) proceeds as follows. An individual L applied to an input image s builds a hierarchy of primitive groups in P. Each invoked tree node creates a new group of primitives and groups of primitives from the lower levels of the hierarchy. In

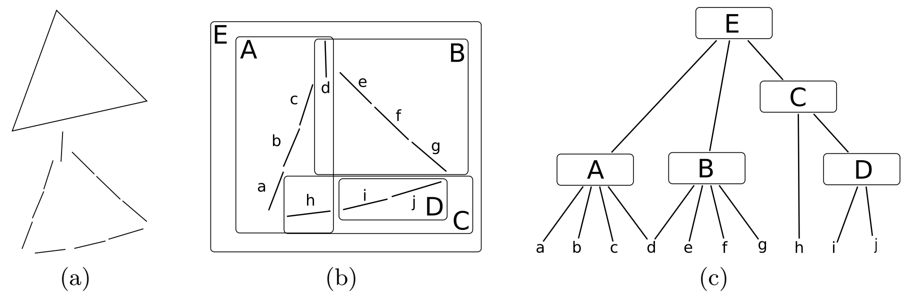


Fig. 1. A simple geometric figure s and its primitive representation P (a); the primitive hierarchy built by the learner from primitives, imposed on the image (b) and shown in an abstract form (c).

Table 1. The genetic operators (function set).

Type	Operator
$\Re$	ERC – Ephemeral Random Constant
Ω	*Input*() – the primitive representation P of the input image s
A	p_x, p_y, p_o – the attributes of visual primitives
R	*Equals*, *Equals5Percent*, *Equals10Percent*, *Equals20Percent*, *LessThan*, *GreaterThan*
G	*Sum*, *Mean*, *Product*, *Median*, *Min*, *Max*, *Range*
$\Re$	+($\Re$,$\Re$), –($\Re$,$\Re$), *($\Re$,$\Re$), /($\Re$,$\Re$), *sin*($\Re$), *cos*($\Re$), *abs*($\Re$), *sqrt*($\Re$), *sgn*($\Re$), *ln*($\Re$), *AttributeValue*(Ω,$\Re$)
Ω	*SetIntersection*(Ω,Ω), *SetUnion*(Ω,Ω), *SetMinus*(Ω,Ω), *SetMinusSym*(Ω,Ω), *SelectorMax*(Ω,A), *SelectorMin*(Ω,A), *SelectorCompare*(Ω,A,R,$\Re$), *SelectorCompareAggreg*(Ω,A,R,G), *CreatePair*(Ω,Ω), *CreatePairD*(Ω,Ω), *ForEach*(Ω,Ω), *ForEachCreatePair*(Ω,Ω,Ω), *ForEachCreatePairD*(Ω,Ω,Ω), *Ungroup*(Ω), *GroupHierarchyCount*(Ω,$\Re$), *GroupHierarchyDistance*(Ω, $\Re$), *GroupProximity*(Ω, $\Re$), *GroupOrientationMulti*(Ω, $\Re$), *Draw*(Ω)

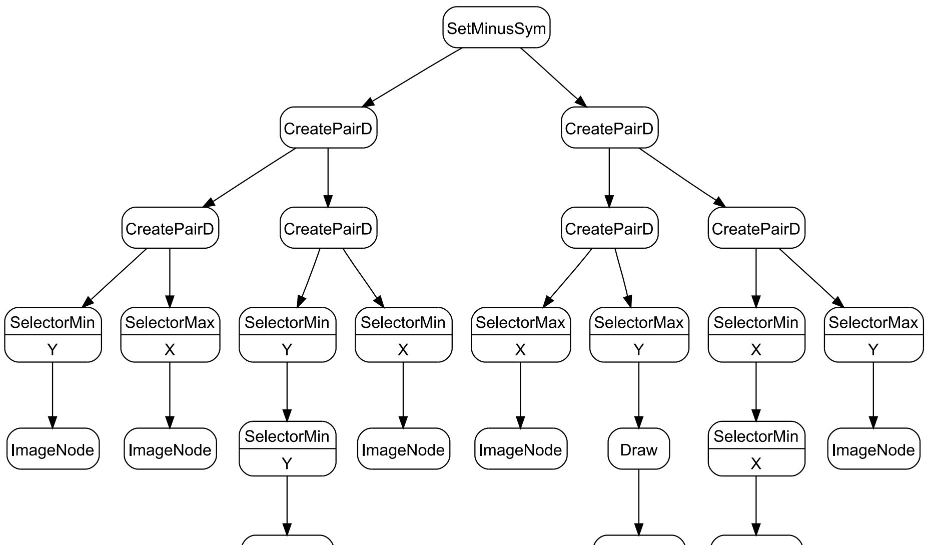


Fig. 2. An exemplary evolved individual capable of recognizing triangular shapes.

the end, the root node returns a nested primitive hierarchy built atop of P, which reflects the processing performed by L for s.

An exemplary process of recognition by a hypothetical learner is shown in Fig. 1. Figure 1b depicts the hierarchical grouping imposed by L on P in the process of recognition of image from Fig. 1a (procedure GroupAndDraw). Figure 1c presents the same hierarchy in abstraction from the input image. Note that the groups of primitives are allowed to intersect.

Internally, an individual is a procedure written in a form of a tree, with internal nodes representing *functions* that process sets of primitives. Every visual primitive has three attributes: the coordinates of its midpoint (p_x, p_y), and the orientation p_o expressed as an absolute angle with respect to abscissa. The terminal nodes fetch the initial set of primitives P derived from the input image s, and the consecutive internal nodes process the primitives, all the way up to the root node. The functions, presented in Table 1, may be divided into scalar functions, selectors, iterators, and grouping operators. Scalar functions implement conventional arithmetic. Selectors filter primitives based on their attributes; for instance, calling *LessThan*$(A_1, p_x, 20)$ returns only those primitives from the set A_1 that have $p_x < 20$. Iterators process primitives one by one; e.g., *ForEach*(A_1, A_2) will iterate over all primitives from the set A_1, process every one of them independently using the program (subtree) defined by its second argument A_2, gather the results of that processing into a single set, and return that set. Grouping operators group primitives based on their attributes and features; e.g., *GroupProximity*$(A_1, 10)$ will group the primitives in A_1 according to their spatial proximity, using the 10 argument as a threshold for proximity relation, and return the results of that grouping as a nested hierarchy of sets (every results of type Ω (see Table 1) is in general a hierarchy of sets of primitives, as shown in example in Fig. 1).

Since the above instructions operate on different types, we use strongly-typed genetic programming (Montana, 1993), which implies that two nodes may be connected only if their input/output types match. The list of types includes numerical scalars, sets of primitives, attribute labels, binary arithmetic relations, and aggregators.

Fig. 2 presents an exemplary evolved individual capable of recognizing triangular shapes. Let us clarify the recognition process by tracking the execution of the leftmost tree branch (bottom-up processing). The *SelectorMin* operator selects from the entire input set of primitives (supplied by the *ImageNode*) those that have minimal value of the Y coordinate. The resulting group of primitives is merged by operator *CreatePairD* with the primitives selected by the sibling branch (nodes *ImageNode* and *SelectorMax*). That group of primitives is merged again by subsequent *CreatePairD* node with another group of primitives. Finally, the resulting group of primitives is processed by symmetric set difference (*SetMinusSym*) at the root node.

Apart from grouping of visual primitives, some branches of the tree may contain drawing instructions that affect the canvas, which is ultimately subject to evaluation by the Similarity function in Alg. 1. Drawing actions insert line segments into the output canvas c. The coordinates of line segments are derived from the processed primitives. The simplest representative of drawing nodes is the function called *Draw*, which expects primitive set T as an argument and draws on canvas c line segments connecting each pair of primitives from T. *Draw* does not modify the set of primitives it processes. Other drawing functions, with names ending with upper-case letter D (see Table 1), perform drawing as a side-effect of their processing.

The recognition of image s by an individual L is completed by comparing the contents of canvas c to the original image s (the Similarity function in Alg. 1). It is assumed that the difference between c and s is proportional to the minimal total cost of bijective assignment of pixels lit in c to pixels lit in s. The total cost is a sum of costs for each pixel assignment. The cost of assignment depends on the distance between pixels: when the distance is less than 5, the cost is 0; maximum cost 1 is assigned when the distance is greater than 15; between 5 and 15 the cost is a linear function of the distance. These thresholds were adjusted by experimentation and depend on the lower limit of distance between primitives specified in the preprocessing procedure (see (Jaśkowski *et al.*, 2007c) for details). For each pixel that cannot be paired (e.g., because there are more lit pixels in c than in s), an additional penalty of value 1 is added to the total cost. In such a way, an individual is penalized for committing both false negative errors (when parts of input shape are not drawn on the canvas) and false positive errors (the excess of drawing), including the special case of drawing line segments that overlap (partially or completely). The pairing of image and canvas pixels is carried out by an effective greedy heuristic. The heuristic iterates over canvas pixels and to each of them assigns the closest non-paired pixel from the input image.

To evaluate the fitness of an individual-learner L, the above recognition process is carried out for all training images from the training set S of images. The (minimized) fitness of L is defined as the total

Algorithm 2 Evolution of classifiers for a single decision class. I^1 is the initial population, g_{max} is the number of generations, and O stands for the set of breeding operators.

```
1: function EVOLVE($I^1$, $g_{max}$, $O$)
2:   for $g \leftarrow 1 \dots g_{max} - 1$ do
3:     Evaluate each $L \in I^g$
4:     $B \leftarrow$ SELECTION($I^g$)
5:     $I^{g+1} \leftarrow$ apply breeding operators
           from $O$ to $B$
6:   end for
7:   return $I^{g_{max}}$
8: end function
```

Algorithm 3 GPCR evolutionary process. m is the length of the primary run, n is the length of entire evolutionary process (primary run and secondary run).

```
1: function GPCR($m$, $n$)                          ▷ $m < n$
2:   for $c \leftarrow 1 \dots k$ do        ▷ Loop over primary runs
3:     $I_c^1 \leftarrow$ RANDOMPOPULATION( )
4:     $P_c \leftarrow$ EVOLVE($I_c^1$, $m$,
         {MUTATE, CROSSOVER})
5:   end for
6:   $P \leftarrow (P_1, \dots, P_k)$         ▷ Pools for crossbreeding
7:   for $c \leftarrow 1 \dots k$ do  ▷ Loop over secondary runs
8:     $I_c^n \leftarrow$ EVOLVE($I_c^1$, $n - m$,
         {MUTATE,CROSSOVER, CROSSBREED($P$)})
9:   end for
10:  return $(I_1^n, \dots, I_k^n)$
11: end function
```

cost of the assignment normalized by the number of lit pixels in s, and averaged over the entire training set of images S. An ideal individual perfectly reproduces shapes in all training images using the minimal number of line segments, so that its fitness amounts to 0. The more the canvas produced by L differs from the input image (for one or more training images), the greater (worse) its fitness value. Thus, the fitness function rewards individuals that exactly and completely reproduce as many images from S as possible, promoting so the discovery of similarities between the training images.

Let us now explain how the generative recognition process is employed in the multi-class classification task that is of interest in this paper. For each decision class (handwritten character class in our experiment), we run a separate evolutionary process that uses only examples from that class in the training set S. A highly fit individual resulting from such a process should be able to well reproduce the shapes represented by images in S, and at the same time is unlikely to accurately (and minimally) reproduce the shapes from the other classes. Examples of such close-to-perfect and imperfect reproductions will be given in Figs. 6a and 6b in the experimental part of this paper.

In other words, our learning algorithm uses training examples from the positive class only, having no idea about the existence of other classes (object shapes). This learning paradigm, known as *one-class learning* (Moya and Hostetler, 1993), may be beneficial in terms of training time (fewer training examples) and is context-free in the sense that no other classes are involved in the training process. To handle a k-class classification problem, we run k independent evolutionary processes, each of them devoted to one class (see Alg. 2). The k best individuals obtained from particular runs form the complete multi-class classifier (*recognition system*), ready to recognize new images using a straightforward procedure detailed in Section 5.2.

4. Code Reuse

Given the similar visual nature of the learning tasks related to particular classes of characters, we expect them to require some common knowledge. Some classes may need similar fragments of genetic code to, e.g., detect the important features like stroke ends or stroke junctions. For instance, locating the ends of the shape of letter Y may require a similar code (subtree) as locating the ends of letter X. To exploit such commonalities and avoid unnecessary redundancy, we enable cross-task code reuse between the evolutionary processes devoted to particular classes.

The method, *Genetic Programming with Code Reuse* (GPCR, Alg. 3), runs in parallel k evolutionary processes for n generations, one process for each of k classes. For the initial m generations ($m < n$), evolution proceeds exactly as in the basic algorithm described in the previous section (referred to as GP in the following). We call this part of evolutionary process the *primary run* (line 4 in Alg. 3). When the c^{th} run ($c = 1...k$) reaches the m^{th} generation, we store a snapshot (copy) of its population in a *pool* P_c. Next, the process's population is re-initialized in the same way as the initial population of the primary run, and the evolution continues for the remaining $n-m$ generations, referred to as *secondary run* (line 8 in Alg. 3). The secondary run is initialized using the same random seed as the primary run, so the initial populations of the primary and secondary runs are exactly the same (I^1 in Alg. 3).

The secondary runs differ from the primary ones in that they activate an extra *crossbreeding operator*

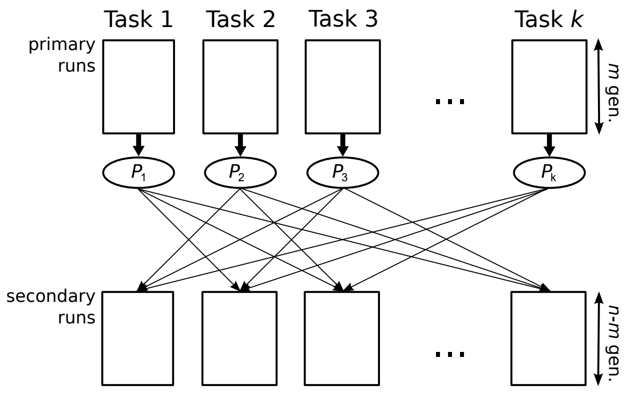


Fig. 3. The architecture of GPCR.

that is allowed to import genetic material from the pools. Its algorithm is presented in Alg. 4, along with the standard breeding operators used in genetic programming. Crossbreeding for the c^{th} secondary run works similarly to subtree-swapping crossover, however, it interbreeds an individual from the current population (a 'native') with an individual from one of the pools of the other decision classes P_j, $j \neq c$ (an 'alien'). First, it selects a native parent from the current population using the same selection procedure as crossover. Then, it picks out an alien parent by randomly choosing one of the pools P_j, $j \neq c$, and then randomly selecting an individual from P_j, disregarding its fitness. Fitness is ignored in this process as it reflects alien's performance on a different task (decision class); in particular, alien's low fitness does not necessarily mean that it lacks code fragments that are useful for solving the native's task. Finally, the crossbreeding operator randomly selects two nodes N_n and N_a in the native and the alien parent respectively, and replaces N_n by the subtree rooted in N_a. The modified native parent (offspring) is injected into the subsequent population (provided it meets the tree depth constraint), and the alien parent is discarded. Thus, crossbreeding may involve large portions of code as well as small code fragments, in an extreme case even single terminal nodes.

Fig. 3 outlines the dataflow in GPCR. Each column of two blocks in the diagram comprises the primary run and the secondary run for one decision class (task), and the arrows depict the transfer of genetic material between them. Overall, GPCR involves k primary runs lasting m generations and k secondary runs lasting $n-m$ generations, so the total number of individual evaluations is the same as in standard GP running k runs for n generations each. Thus, the additional computational overhead brought by GPCR includes the cost of re-initialization of k populations at the verge between the primary and secondary runs

Algorithm 4 Genetic Programming operators

```
function MUTATE($p$)                                  ▷ A single parent
    $s_1 \leftarrow$ randomly selected subtree in parent $p$
    $s_2 \leftarrow$ randomly generated subtree
    return child produced by replacing $s_1$ with
            $s_2$ in $p$
end function

function CROSSOVER($p_1$, $p_2$)                       ▷ Two parents
    $s_1 \leftarrow$ randomly selected subtree in parent $p_1$
    $s_2 \leftarrow$ randomly selected subtree in parent $p_2$
    $c_1, c_2 \leftarrow$ swap $s_1$ with $s_2$ in $p_1$ and $p_2$
    return $\{c_1, c_2\}$
end function

function CROSSBREED($p$, $c$, $P$)     ▷ A parent, its
  task id, and pools from primary run
    $P_j \leftarrow$ randomly selected pool from $P$
            such that $j \neq c$
    $a \leftarrow$ randomly selected individual from $P_j$
    $s_1 \leftarrow$ randomly selected subtree in parent $p$
    $s_2 \leftarrow$ randomly selected subtree in alien $a$
    return child produced by replacing $s_1$ with
            $s_2$ in $p$
end function
```

and the cost of crossbreeding. As these operations are generally computationally inexpensive when compared to the fitness assessment, the overall computational effort of GPCR is roughly the same as that of GP[1]. This enables fair comparison and allows us to focus exclusively on the performance of the evolved solutions in the experimental part of the paper.

5. The Experiment

In the experimental part, we approach a real-world multi-class problem of handwritten character recognition and demonstrate how GPCR compares to GP on that task in terms of fitness as well as classification accuracy (recognition performance). The task is to recognize letters from Elder Futhark, the oldest form of the runic alphabet. To make the task realistic and self-contained, we consider *all* character classes present in this problem, which look in print as follows:

ᚠ ᚢ ᚦ ᚨ ᚱ ᚲ ᚷ ᚹ ᚺ ᚾ ᛁ ᛃ
ᛇ ᛈ ᛉ ᛊ ᛏ ᛒ ᛖ ᛗ ᛚ ᛜ ᛞ ᛟ

Elder Futhark letters are written with straight pen strokes only, which makes them a good testbed

[1]The cost of population re-initialization is in fact close to zero: as the secondary run starts with exactly the same population as the primary run, one can cheaply create a copy of the initial population of the primary run and use it as a starting point for the secondary run.

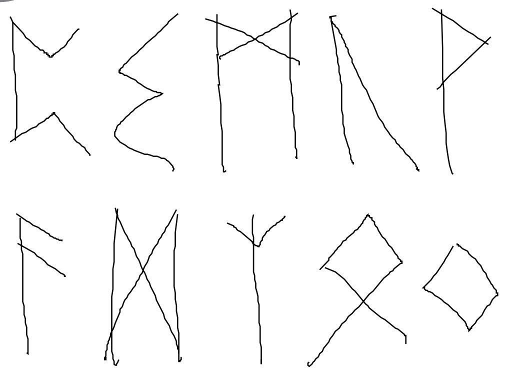


Fig. 4. Examples of handwritten Elder Futhark characters.

for our approach that uses line segments as drawing actions (see Section 3). Using a TabletPC computer, we prepared the training set containing 240 images (examples, objects) of $k = 24$ runic alphabet characters, each character class represented by 10 examples written by 7 persons (three persons provided two character sets each). Fig. 4 shows examples of selected handwritten characters.

The purpose of the experiment is to compare the method with code reuse (GPCR) to the basic genetic programming (GP) that provides us with the control results. Technically, for each of $k = 24$ character classes we run a generational evolutionary process maintaining a population of 10,000 individuals. Each process lasts for $n = 600$ generations; this number of generations was necessary for GP to reach a firm leveling-off of fitness graph indicating little chance for further improvement. To provide for statistical significance, we repeat each process 30 times, starting each time from a different initial population created using Koza's ramped half-and-half operator with ramp from 2 to 6 (Koza, 1992). We apply the tournament selection with tournament of size 5, using individuals' sizes for tie breaking and thus promoting smaller trees and alleviating the problem of code bloat. The tree depth limit is set to 10; the mutation and crossover operations may be repeated up to 5 times if the resulting individuals do not meet this constraint; otherwise, the parent solutions are copied into the subsequent generation. The algorithm was implemented in Java with help of the ECJ package (Luke, 2002), except for the fitness function written for efficiency in C++. For the evolutionary parameters not mentioned here, ECJ's defaults were used.

For GP runs, offspring are created by crossing over the selected parent solutions from the previous generation (with probability 0.8; see CROSSOVER function in Alg. 4), or mutating them by replacing subtrees (with probability 0.2; see MUTATE function in Alg. 4). For GPCR, the primary run lasts for $m = 300$ generations with the same settings as GP and the same contents of the initial population, so it is literally the same as the first m generations of the corresponding GP run. In the secondary run (lasting for 300 generations as well) the mutation probability is lowered to 0.1 and the remaining 0.1 is yielded to the crossbreeding operator. Apart from this shift in probabilities associated with particular operators, the GP and GPCR settings are virtually identical and take the same computation time.

To intensify the search and prevent premature convergence to local optima, we used the island model (Whitley *et al.*, 1999). We split the population into 10 equally-sized islands and, starting from the 50th generation, exchange individuals between the islands every 20th generation. During the exchange, every odd-numbered island donates the clones of its best-performing individuals (10% of population, selected by tournament of size 5) to the five even-numbered islands, where they replace the worst-performing individuals selected using an inverse tournament of the same size. Reciprocally, the even-numbered islands donate their representatives to the odd-numbered islands in the same way. The islands should *not* be confused with the boxes depicting the evolutionary runs in Fig. 3 — the island model is implemented within *each* evolutionary run independently, both in GPCR and in basic GP, so its presence does not bias their comparison.

5.1. Fitness comparison. In order to confront GPCR with GP, we first compared their results in terms of fitness. Table 2 presents the average fitness (computed on the training set) of the best-of-run individuals for each runic letter. Clearly, the fitness for GPCR is better in all cases. Statistical analysis reveals that, for 21 classes marked by stars, reusing the code significantly pays off (t-test, $\alpha = 0.05$).

We also evaluated the best-of-run individuals on the test set of characters, which was disjoint with the training set and contained 1440 images, that is 60 images for each character class. On the test set, GPCR outperformed GP for 22 classes, and in 15 cases the difference was statistically significant (see Table 3). The two cases where GP outdid GPCR were statistically insignificant. On average, the fitness of the GPCR best-of-run individual was better than that of the GP best-of-run individual by nearly 21% for the training set and by nearly 24% on the test set.

Figure 5 presents the progress of GP (solid lines) and GPCR evolutionary processes (dotted lines) for the 5 first letters of the runic alphabet. These graphs are representative for the behavior of both methods

Table 2. Fitness of best-of-run individuals for the training set for GPCR and GP averaged over 30 runs for each runic letter. The lower the value, the better fitness; stars mark statistical significance (t-test, $\alpha = 0.05$).

Letter	ᚠ	ᚢ	ᚦ	ᚨ	ᚱ	ᚲ	ᚷ	ᚹ	ᚺ	ᚾ	ᛁ	ᛃ
GP	0.301	0.128	0.275	0.261	0.261	0.226	0.212	0.185	0.166	0.153	0.066	0.235
GPCR	0.188*	0.119*	0.144*	0.205*	0.196*	0.191*	0.192*	0.173*	0.149*	0.149*	0.066	0.217*
Letter	ᛇ	ᛈ	ᛉ	ᛊ	ᛏ	ᛒ	ᛖ	ᛗ	ᛚ	ᛜ	ᛞ	ᛟ
GP	0.214	0.447	0.203	0.299	0.214	0.293	0.208	0.249	0.182	0.244	0.235	0.278
GPCR	0.197	0.267*	0.188*	0.254*	0.211*	0.226*	0.184*	0.187*	0.164*	0.215	0.212*	0.257*

Table 3. Overall comparison of GPCR and GP best-of-run fitness. GPCR is better for all 24 letters on the training set and for 22 letters on the test set. These differences are statistically significant in 21 cases in training and 15 cases in testing (t-test, $\alpha = 0.05$).

GPCR	Average fitness gain	# better	# worse	# stat. better	# stat. worse
Training set	20.9%	24	0	21	0
Test set	23.9%	22	2	15	0

for all 24 characters. GPCR series start from the 300^{th} generation, when the secondary run is initialized, while its primary runs are the same as GP's first 300 generations. The convergence of GPCR is much faster than GP's: the line is nearly vertical for the first few generations of the secondary run. Note also that GPCR significantly outperforms GP on the sophisticated characters, while on a simple one (ᚢ) it does not have much chance for improvement.

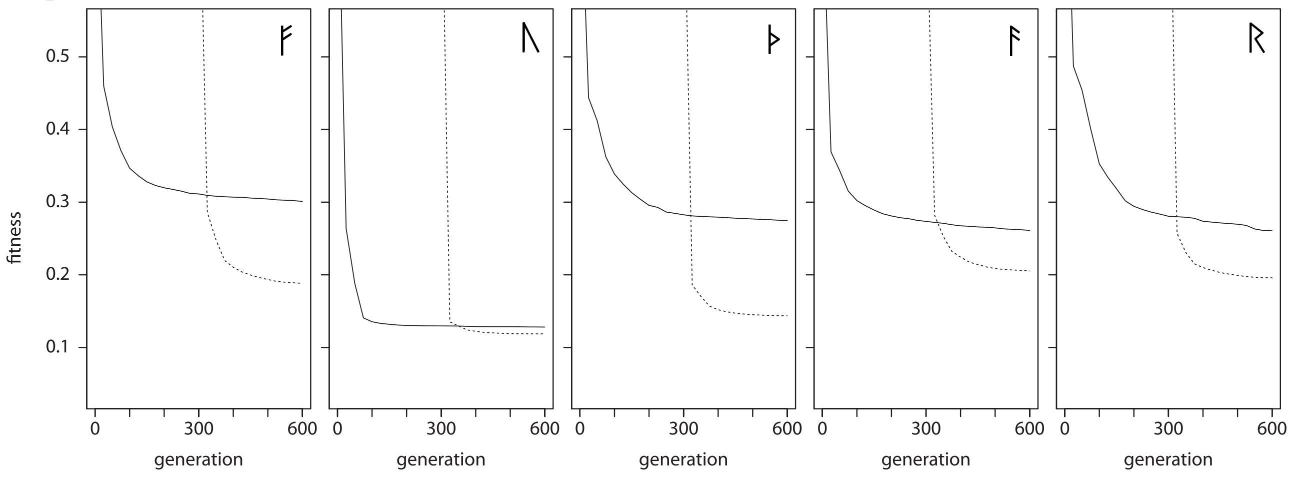
5.2. Comparison of recognition systems.

We compare GPCR and GP also in terms of the multi-class recognition performance. For this purpose, we combine all 24 best-of-run individuals, forming the complete *recognition system* that undergoes evaluation on the test set of characters. The system classifies an example t by computing fitness values of all 24 individuals for t and indicating the class associated with the individual that produced the smallest (the best) value. Such procedure is motivated by an observation, that individual's fitness should be close to 0 only for the positive examples (images from the class it was trained on). For the negative examples, the reproduction process should most probably fail, drawing inappropriate line segments on the canvas and thus resulting in inferior fitness (cf. Section 3). As a demonstration, Fig. 6a presents the close-to-perfect reconstructions produced by the 24 best-of-run individuals for examples from the corresponding positive classes. On the other hand, in Fig. 6b, where each shape was reconstructed using the same individual taught on class ᛇ, only the reconstruction of character ᛇ is correct[2].

The above recognition procedure, called *simple recognition system* in the following, may be obtained at a relatively low computational expense of one evolutionary process per class. Given more runs, recognition accuracy may be further boosted by employing more *voters* per each class, as opposed to one-voter-per-class scheme used by the simple recognition system. This is especially appealing in the context of evolutionary computation, where each run usually produces a different best-of-run individual, so their fusion may be synergistic. To exploit this opportunity, we come up with a *vote-l* recognition system that uses l best-of-run individuals for each class. Given 30 runs performed for each class in this study, we build $\lfloor 30/l \rfloor$ separate vote-l systems. When classifying an example, the vote-l system considers all l^{24} possible combinations of voters, using one voter per class in each combination. Each combination produces one decision, using the same procedure as the simple recognition system. The class indicated most frequently across all combinations is the final decision of the vote-l recognition system. Technically, there is no need to construct all l^{24} ensembles, because the result can be computed at lower than quadratic time complexity.

Table 4 compares GP to GPCR with respect to the test-set classification accuracy of the simple recognition system and the vote-l method with a different number of voters l. The table shows the averages

[2]Though also reconstruction of letter ᛁ seems correct, closer examination reveals surplus overlying strokes that will be penalized by the fitness measure.



(a) Training set

Fig. 5. Fitness graphs of best-so-far individuals for the 5 first letters of runic alphabet (ᚠ, ᚢ, ᚦ, ᚨ and ᚱ). Solid line corresponds to the GP experiment whereas dotted line to the GPCR one. Averaged over 30 evolutionary processes.

Table 4. Test-set classification accuracy (%) for different voting methods.

	simple	vote-2	vote-3	vote-4	vote-5	vote-30
GP	69.79±1.66	78.50±1.12	82.50±1.02	85.21±0.79	86.66±0.61	91.32
GPCR	81.94±0.89	87.88±0.49	91.19±0.41	92.58±0.31	93.18±0.27	95.56

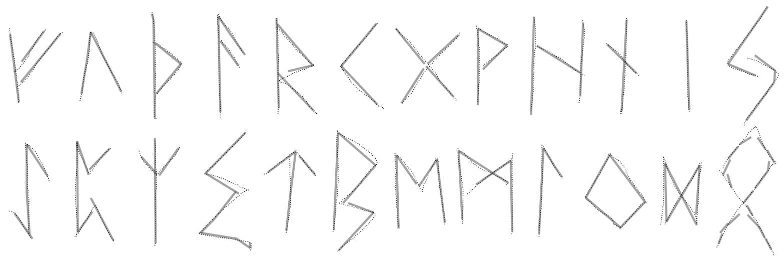
with .95 confidence intervals; for vote-30, confidence intervals cannot be provided as, with 30 independent runs for each character class, only one unique vote-30 recognition system can be built. Quite obviously, the more voters, the better the performance. But more importantly, GPCR improves the baseline result of GP for every number of voters. The gap between them narrows when the number of voters increases, but remains substantial even for the vote-30 case.

Finally, these results are also competitive to the commonly used standard techniques. A radial basis function neural network, a support vector machine, and a 3-layer perceptron, when taught on the same training data digitized to 32 × 32 raster images, attain test-set accuracy of 79.25%, 89.75%, and 90.67%, respectively (using the default parameter settings provided by the Weka software environment (Hall *et al.*, 2009)).

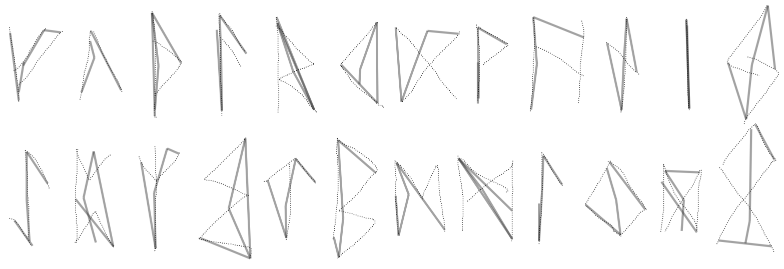
Table 5 presents the test set results for the vote-30 GPCR experiment in terms of true positives and false positives. Nine out of 24 characters are recognized perfectly. Overall, only three characters are recognized in less than 90% of cases: ᚢ, ᚹ, and ᛚ. Analysis of the confusion matrix (not shown here for brevity) allows us to conclude that ᚢ is sometimes mistaken for ᛚ, hence both yield high false positive errors. Similarly, the recognition system occasionally incorrectly classifies ᚹ as ᚦ. Since the letters in these pairs are very similar to each other and even a human may find it difficult to tell them apart in handwriting, this result may be considered appealing.

5.3. Discussion. We demonstrated that code reuse between different tasks may boost the results of the evolutionary process. Although we considered a classification problem here, it is important to notice that GPCR and GP were compared, above all, on the set of *optimization* tasks (Section 5.1) with a well defined, close-to-continuous fitness function (described in Section 3). This suggests that applicability of cross-task code reuse is not limited exclusively to machine learning and pattern recognition. We hypothesize that other types of tasks, like regression, could benefit from such code reuse as well, provided the existence of common knowledge that helps solving multiple tasks.

On the other hand, the results suggest that using a generative, close-to-continuous fitness function based on *one-class* data to evolve components of a complex recognition system that makes discrete decisions about *multiple* classes was a good choice. In particular, the fitter individuals obtained in GPCR translated into the better classification performance of the recognition system, which we shown in Section 5.2.



(a)



(b)

Fig. 6. (a) Individuals tested on examples from their positive classes produce high-quality reproductions. (b) An individual taught on class ᛇ, when tested on examples from all classes, produces poor reproduction for most of the negative examples. Dotted line – input image, continuous line – reconstruction (drawing actions).

Table 5. True positive (TP) and false positive (FP) ratios for vote-30 GPCR method.

Letter	ᚠ	ᚢ	ᚦ	ᚨ	ᚱ	ᚲ	ᚷ	ᚹ	ᚺ	ᚾ	ᛁ	ᛃ
TP	90.0%	80.0%	100%	95.0%	98.3%	100%	98.3%	81.7%	100%	93.3%	98.3%	98.3%
FP	0.0%	18.3%	21.7%	1.7%	0.0%	1.7%	3.3%	3.3%	0.0%	6.7%	8.3%	0.0%
Letter	ᛇ	ᛈ	ᛉ	ᛊ	ᛏ	ᛒ	ᛖ	ᛗ	ᛚ	ᛜ	ᛞ	ᛟ
TP	100%	98.3%	93.3%	100%	95.0%	100%	100%	96.7%	80.0%	100%	100%	96.7%
FP	3.3%	1.7%	0.0%	1.7%	3.3%	0.0%	3.3%	0.0%	21.7%	6.7%	0.0%	0.0%

Though the reader may be tempted to consider GPCR (and other methods that transfer knowledge between different tasks) as yet another performance improvement technique, there is something more to it. Cross-task reuse of code cannot take place in isolation: learning/optimization process requires an *external* knowledge source that has not been derived from the same training data. In this sense, the setup of GPCR is different from the setup of performance improvement techniques that do not require such measures, like automatically defined functions. In return, GPCR enables in-depth analysis of task similarity that is subject of the following section.

GPCR with its crossbreeding operator can be seen as a variant of island model in which the individuals on different islands learn different tasks and optimize different fitness functions. Note, however, that in the island model genetic material is exchanged, usually, every several generations. In contrast, in GPCR genetic material is exchanged between the evolutionary processes ("islands") only once: at the end of the primary run. The crossbreeding operator activated in the secondary runs uses this once-saved genetic material over and over again, but never modifies it anymore. Though it is possible to modify GPCR to resemble the island model more, here we chose a setup that was conceptually less complex and technically easier to implement in our computational infrastructure (less communication between "islands").

6. Detailed analysis of code reuse

The results presented so far reveal the 'symptoms' of code reuse, i.e., its impact on individuals' performance. In the following we look under the hood of that process, attempting to describe it in a more quantitative manner by 'data mining' a secondary run. In particular, the questions of interest are i) how much of individual's knowledge results from crossbreeding, mutation and crossover; and ii) how much knowledge is adopted by an individual from a particular pool P_i.

Since an individual is a tree-like expression, two types of approaches may be considered: *node-oriented* and *edge-oriented.* The idea behind the node-oriented analysis is to count the tree nodes in individuals and distinguish the *initial-nodes*, the *mutation-nodes* and the *pool-nodes*, depending on node's origin, i.e., the initial population, mutation, or crossbreeding. However, such node-oriented approach would quantify *information* rather than knowledge, since it would not take into account the mutual connections of tree nodes. That is why we favor here an edge-oriented analysis and define the *initial-edges*, the *mutation-edges* and the *pool-edges*, depending on the source an edge comes from. An edge connecting the parent tree with the subtree inserted by mutation, crossover, or crossbreeding will be called a *new-edge.* For any individual, the initial-edges, the mutation-edges, the pool-edges, and the new-edges sum up to the total number of edges.

We define the amount of knowledge received by an individual from the primary run as the total number of its pool-edges. For an individual composed of nodes that come each from a *different* genetic operation, this measure amounts to 0, no matter what the actual numbers of pool-nodes is. This is consistent with intuition: such a tree does not inherit even a single pair of connected nodes from the primary run, so no code is reused (the arrangement of nodes could as well result from mutation). Moreover, unlike the node-oriented measure, the edge-oriented measure prefers one subtree of $n_1 + n_2$ nodes reused from pool P_i to two separate subtrees of sizes n_1 and n_2 also reused from P_i. Such behavior is reasonable, since in the latter case, the total amount of knowledge is reduced due to partitioning of the genetic code.

We chose the letter ᚦ for the detailed analysis of code reuse to find out why the results of GPCR for this class are much better than the results of GP. Table 6 presents the statistics of edge types for the best-of-run individuals, averaged over the 30 runs. Our first observation is that nearly half of the edges (48.5%) comes from the pools whereas the rest (51.5%) results from the other genetic operators (population initialization, mutation, and crossover). Only 7.7% of the code comes from mutation despite the fact that its probability is the same as probability of crossbreeding (0.1). This confirms that crossbreeding is a valuable source of useful pieces of genetic code. Apparently, the trees in the pools perform useful computations, so they are preferred to the purely random subtrees inserted by mutation.

Table 7 presents the sorted distribution of edges reused from particular pools by the best-of-run individuals for class ᚦ. It should not come as a surprise that pool-ᚹ, amounting to more than 9% of reused code (11.2 edges per individual on average), helped the most to recognize the examples from the class ᚦ (see Table 7). As no other letter is as visually similar to ᚦ as ᚹ, evolution reused the solutions for ᚹ. Their code fragments proliferated throughout the population and increased the survival chances of the crossbred individuals. We hypothesize that this is also why it was so easy for GPCR to beat GP on this letter.

It is striking how the top five ranked letters (ᚹ, ᚠ, ᚨ, ᛇ, ᛚ) are similar to ᚦ in shape — they all have one vertical segment and at least one diagonal segment on the right. On the other hand, the last entries of Table 7 contain mostly the letters that are visually dissimilar to ᚦ — with multiple long diagonal or vertical lines or without long lines at all. The least used pool corresponds to the simplest letter ᛁ, which was probably not challenging enough to give rise to a reusable code.

Figure 7 shows the dynamics of the edge type statistics during the evolution for letter ᚦ, averaged over the 30 runs. Only the first 100 generations of the secondary run are shown since the graph does not change much after this point. We can observe that the total number of pool edges grows very fast in the beginning of the evolution, reaches the peak value of 85 around the 15^{th} generation, and then, after dropping a bit, oscillates between 65 and 75. During the first 30 generations some interesting patterns emerge. Firstly, we observe the steady growth of edges from pool-ᚹ that produces the widest stripe in the end of graph, which is in accordance with the superior utility of pool-ᚹ code for ᚦ, discovered in the static analysis. Secondly, the drop of the total number of reused edges from 85 edges in the 15^{th} generation to 65 edges in the 50^{th} generation is mostly caused by removing the edges previously acquired from pool-ᚠ and (less prominently) from pool-ᚨ. We hypothesize that the code acquired from these pools, initially very useful, was later replaced by the code from pool-ᚹ and by the new-edges, since the number of new-edges grows monotonically during the first 100 generations.

7. Conclusions

Despite the large number of classes (24), the low number of training examples per class (10), the variability of handwriting styles (7 persons), and without access to negative examples (one-class learning), our evolutionary learning method attains an attractive classification accuracy on the large and diversified test set in a real-world problem of handwritten character recognition. Most of the errors committed by the GPCR recognition systems involve character classes that are

Table 6. The distribution of edge types for the best-of-run individuals for class ᚦ averaged over 30 runs. The average total number of edges is 124 (100%).

Edge type	initial-edges	mutation-edges	pool-edges	new-edges
edges	0.9	9.6	60.1	53.5
%	0.7%	7.7%	48.5%	43.1%

Table 7. The sorted distribution of edges reused from particular pools by the best-of-run individuals for class ᚦ (averaged over 30 runs).

Letter	ᚹ	ᚠ	ᚨ	ᛇ	ᛚ	ᛊ	ᚱ	ᛈ	ᛒ	ᛏ	ᛗ	ᚢ
pool-edges	**11.2**	4.1	3.8	3.6	3.2	3.1	3.1	3.1	3.1	3.0	2.5	2.2
%	**9.1%**	3.3%	3.1%	2.9%	2.6%	2.5%	2.5%	2.5%	2.5%	2.4%	2.0%	1.8%
Letter	ᛞ	ᛜ	ᚺ	ᛉ	ᛃ	ᚲ	ᚾ	ᚷ	ᛖ	ᛟ	ᛁ	
pool-edges	2.1	2.0	1.6	1.5	1.4	1.3	1.2	0.9	0.9	0.7	0.4	
%	1.7%	1.6%	1.3%	1.2%	1.2%	1.0%	1.0%	0.8%	0.7%	0.6%	0.3%	

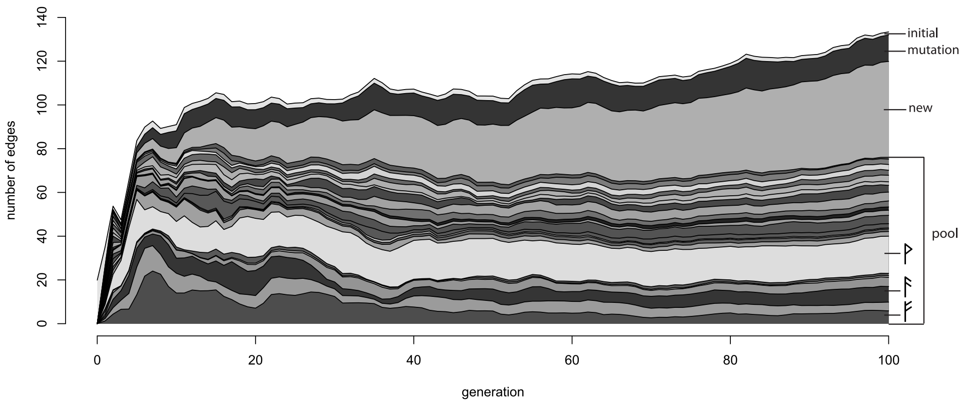


Fig. 7. The numbers of different edge types in individuals in the first 100 generations of secondary run of letter ᚦ. Each stripe represents one edge type. The graphs were averaged over 100 best-of-generation individuals and 30 evolutionary runs.

hard to tell apart also for humans (e.g., ᚢ and ᛚ, ᚹ and ᚦ). Thanks to one-class learning, the recognition system may be easily extended by a new class by running a separate evolutionary process; the existing components do not have to be modified.

GPCR, the proposed mechanism of code reuse, boosts fitness and recognition accuracy, so that the resulting recognition system outperforms the standard GP. GPCR is a straightforward, easy-to-implement extension of the canonical genetic programming, and its extra computational overhead is close to insignificant. It may be hypothesized that it would prove helpful also in other recognition tasks where common visual primitives occur in many decision classes, like handwriting recognition for contemporary languages (e.g., Chang *et al.* (2010b) and Ciresan *et al.* (2012)). Moreover, the results in terms of fitness suggest that GPCR would be beneficial also for other types of tasks that do not necessarily end up with a multi-class recognizer.

In a more general perspective, we demonstrated that the paradigm of genetic programming, thanks to symbolic representation of solutions and the ability of abstraction from a specific context, offers an excellent platform for knowledge transfer. However, there is not enough evidence yet to claim that these results generalize beyond our genetic programming-based visual learning. At least three major factors determine the profitability of code reuse: the representation of solutions, the presence of commonalities between the learning tasks, and the crossbreeding operator. In particular, an appropriate crossbreeding operator that preserves the chunks of code for a specific knowledge representation may improve the convergence of the secondary run. This leads to an interesting idea of posing an inverse problem that may become a further research topic: instead of trying to devise a code reuse method for a particular knowledge representation, one could try to design a knowledge representation that enables code reuse. Furthermore, analogously to evolving evolvability, this could lead to the concept of *evolving reusability*, where the objective for the evolutionary process would be to evolve individuals' encoding that promotes code reusability.

The results obtained in this study suggest a successful match between the abstraction level of knowledge representation and the level at which GPCR operates. Supposedly, our operators that group and select visual primitives are well suited for identifying the common subtasks (code fragments) and their reuse. The detailed analysis shows that knowledge proliferates most successfully throughout the population when transferred from the similar tasks (e.g., characters ᚦ and ᚹ). We hypothesize that, among the reused code fragments, the most useful are those ones that help the individuals to detect object features that repeat across the classes, like stroke junctions and stroke ends. However, this supposition needs a more in-depth analysis and will be subject of another study.

References

Bhanu, B., Lin, Y. and Krawiec, K. (2005). *Evolutionary Synthesis of Pattern Recognition Systems*, Springer-Verlag, New York.

Caruana, R. (1997). Multitask learning, *Mach. Learn.* **28**(1): 41–75.

Chang, Y. F., Lee, J. C., Rijal, O. M. and Bakar, S. A. R. S. A. (2010a). Efficient online handwritten chinese character recognition system using a two-dimensional functional relationship model, *International Journal of Applied Mathematics and Computer Science* **20**(4): 727–738.

Chang, Y., Lee, J., Rijal, O. and Bakar, S. (2010b). Efficient online handwritten chinese character recognition system using a two-dimensional functional relationship model, *Int. J. Appl. Math. Comput. Sci.* **20**(4): 727–738.

Ciresan, D. C., Meier, U. and Schmidhuber, J. (2012). Multi-column deep neural networks for image classification, *CVPR*, IEEE, pp. 3642–3649.

Fabijanska, A. (2012). A survey of subpixel edge detection methods for images of heat-emitting metal specimens, *Applied Mathematics and Computer Science* **22**(3): 695–710.

Galvan Lopez, E., Poli, R. and Coello Coello, C. A. (2004). Reusing code in genetic programming, *in* M. Keijzer, U.-M. O'Reilly, S. M. Lucas, E. Costa and T. Soule (Eds), *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, Vol. 3003 of *LNCS*, Springer-Verlag, Coimbra, Portugal, pp. 359–368.

Ghosn, J. and Bengio, Y. (2000). Bias learning, knowledge sharing, *ijcnn* **01**: 1009.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P. and Witten, I. H. (2009). The weka data mining software: an update, *SIGKDD Explor. Newsl.* **11**(1): 10–18.

Haynes, T. (1997). On-line adaptation of search via knowledge reuse, *in* J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba and R. L. Riolo (Eds), *Genetic Programming 1997: Proceedings of the Second Annual Conference*, Morgan Kaufmann, Stanford University, CA, USA, pp. 156–161.

Holland, J. (1975). *Adaptation in natural and artificial systems*, Vol. 1, University of Michigan Press, Ann Arbor.

Hornby, G. S. and Pollack, J. B. (2002). Creating high-level components with a generative representation for body-brain evolution, *Artif. Life* **8**(3): 223–246.

Howard, D. (2003). Modularization by multi-run frequency driven subtree encapsulation, *in* R. L. Riolo and B. Worzel (Eds), *Genetic Programming Theory and Practice*, Kluwer, chapter 10, pp. 155–172.

Howard, D., Roberts, S. C. and Ryan, C. (2006). Pragmatic genetic programming strategy for the problem of vehicle detection in airborne reconnaissance., *Pattern Recognition Letters* **27**(11): 1275–1288.

Hsu, W. H., Harmon, S. J., Rodriguez, E. and Zhong, C. (2004). Empirical comparison of incremental reuse strategies in genetic programming for keep-away soccer, *in* M. Keijzer (Ed.), *Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference*, Seattle, Washington, USA.

Jaśkowski, W., Krawiec, K. and Wieloch, B. (2007a). Genetic programming for cross-task knowledge sharing, *in* D. Thierens (Ed.), *Genetic and Evolutionary Computation Conference GECCO*, Association for Computing Machinery, pp. 1620–1627.

Jaśkowski, W., Krawiec, K. and Wieloch, B. (2007b). Knowledge reuse in genetic programming applied to visual learning, *in* D. Thierens (Ed.), *Genetic and Evolutionary Computation Conference GECCO*, Association for Computing Machinery, pp. 1790–1797.

Jaśkowski, W., Krawiec, K. and Wieloch, B. (2007c). Learning and recognition of hand-drawn shapes using generative genetic programming, *in* M. G. et al. (Ed.), *EvoWorkshops 2007*, Vol. 4448 of *LNCS*, Springer-Verlag, Berlin Heidelberg, pp. 281–290.

Koza, J. (1992). *Genetic Programming*, MIT Press, Cambridge, MA.

Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, Cambridge Massachusetts.

Koza, J. R., Bennett III, F. H., Andre, D. and Keane, M. A. (1996). Reuse, parameterized reuse, and hierarchical reuse of substructures in evolving electrical circuits using genetic programming, *in* T. H. *et al.* (Ed.), *Proceedings of International Conference on Evolvable Systems: From Biology to Hardware (ICES-96)*, Vol. 1259 of *Lecture Notes in Computer Science*, Springer-Verlag.

Krawiec, K. (2006). Learning high-level visual concepts using attributed primitives and genetic programming, *in* F. R. (Ed.), *EvoWorkshops 2006*, LNCS 3907, Springer-Verlag, Berlin Heidelberg, pp. 515–519.

Krawiec, K. (2007). Generative learning of visual concepts using multiobjective genetic programming, *Pattern Recognition Letters* **28**: 2385–2400.

Krawiec, K. and Bhanu, B. (2005). Visual learning by coevolutionary feature synthesis, *IEEE Transactions on System, Man, and Cybernetics – Part B* **35**(3): 409–425.

Kurashige, K., Fukuda, T. and Hoshino, H. (2003). Reusing primitive and acquired motion knowledge for gait generation of a six-legged robot using genetic programming, *Journal of Intelligent and Robotic Systems* **38**(1): 121–134.

Langdon, W. B. and Poli, R. (2002). *Foundations of Genetic Programming*, Springer-Verlag.

Li, B., Li, X., Mabu, S. and Hirasawa, K. (2012). Towards automatic discovery and reuse of subroutines in variable size genetic network programming, *in* X. Li (Ed.), *Proceedings of the 2012 IEEE Congress on Evolutionary Computation*, Brisbane, Australia, pp. 485–492.

Louis, S. and McDonnell, J. (2004). Learning with case-injected genetic algorithms, *Evolutionary Computation, IEEE Transactions on* **8**(4): 316–328.

Luke, S. (2002). ECJ evolutionary computation system. (http://cs.gmu.edu/ eclab/projects/ecj/).

Mitchell, T. M. (2006). The discipline of machine learning, *Technical Report CMU-ML-06-108*, Machine Learning Department, Carnegie Mellon University.

Montana, D. J. (1993). Strongly typed genetic programming, *BBN Technical Report #7866*, Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA.

Moya, M. R., K. M. W. and Hostetler, L. D. (1993). One-class classifier networks for target recognition applications, *Proceedings world congress on neural networks*, International Neural Network Society, Portland, OR, pp. 797–801.

O'Neill, M. (2009). Riccardo poli, william B. langdon, nicholas F. mcphee: A field guide to genetic programming lulu.com, 2008, 250 pp, ISBN 978-1-4092-0073-4, *Genetic Programming and Evolvable Machines* **10**(2): 229–230.

O'Sullivan, J. and Thrun, S. (1995). A robot that improves its ability to learn.

Perez, C. B. and Olague, G. (2013). Genetic programming as strategy for learning image descriptor operators, *Intelligent Data Analysis* **17**(4): 561–583.

Pratt, L. Y., Mostow, J. and Kamm, C. A. (1991). Direct Transfer of Learned Information among Neural Networks, *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, AAAI, pp. 584–589.

Rizki, M. M., Zmuda, M. A. and Tamburino, L. A. (2002). Evolving pattern recognition systems, *IEEE Transactions on Evolutionary Computation* **6**(6): 594–609.

Roberts, S. C., Howard, D. and Koza, J. R. (2001). Evolving modules in genetic programming by subtree encapsulation, *in* J. F. M. *et al.* (Ed.), *Genetic Programming, Proceedings of EuroGP'2001*, Vol. 2038 of *LNCS*, Springer-Verlag, pp. 160–175.

Rosca, J. P. and Ballard, D. H. (1996). Discovery of subroutines in genetic programming, *in* P. J. Angeline and K. E. Kinnear, Jr. (Eds), *Advances in Genetic Programming 2*, MIT Press, Cambridge, MA, USA, chapter 9, pp. 177–202.

Seront, G. (1995). External concepts reuse in genetic programming, *in* E. V. Siegel and J. R. Koza (Eds), *Working Notes for the AAAI Symposium on Genetic Programming*, AAAI, MIT, Cambridge, MA, USA, pp. 94–98.

Tackett, W. A. (1993). Genetic generation of "dendritic" trees for image classification, *Proceedings of WCNN93*, IEEE Press, pp. IV 646–649.

Teller, A. and Veloso, M. (1997). PADO: A new learning architecture for object recognition, *in* K. Ikeuchi and M. Veloso (Eds), *Symbolic Visual Learning*, Oxford Press, New York, pp. 77–112.

Trujillo, L. and Olague, G. (2006). Synthesis of interest point detectors through genetic programming, *in* M. Keijzer, M. Cattolico, D. Arnold, V. Babovic, C. Blum, P. Bosman, M. V. Butz, C. Coello Coello, D. Dasgupta, S. G. Ficici, J. Foster, A. Hernandez-Aguirre, G. Hornby, H. Lipson, P. McMinn, J. Moore, G. Raidl, F. Rothlauf, C. Ryan and D. Thierens (Eds), *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, Vol. 1, ACM Press, Seattle, Washington, USA, pp. 887–894.

Vilalta, R. and Drissi, Y. (2002). A perspective view and survey of meta-learning., *Artif. Intell. Rev.* **18**(2): 77–95.

Whitley, D., Rana, S. and Heckendorn, R. (1999). The island model genetic algorithm: On separability, population size and convergence, *Journal of Computing and Information Technology* **7**(1): 33–47.