Deep Evolution of Feature Representations for Handwritten Digit Recognition

Alexandros Agapitos, Michael O'Neill, Miguel Nicolau, David Fagan, Ahmed Kattan, Kathleen Curran

***Abstract*—A training protocol for learning deep neural networks, called *greedy layer-wise training*, is applied to the evolution of a hierarchical, feed-forward Genetic Programming based system for feature construction and object recognition. Results on a popular handwritten digit recognition benchmark clearly demonstrate that two layers of feature transformations improves generalisation compared to a single layer. In addition, we show that the proposed system outperforms several standard Genetic Programming systems, which are based on hand-designed features, and use different program representations and fitness functions.**

I. Introduction

Object recognition and image understanding are fundamental tasks of Artificial Intelligence. A great deal of research is devoted to feature engineering for object recognition systems, which is often a tedious process that requires significant human involvement. Instances of state-of-the-art hand-crafted feature descriptors that appear in the literature are gradient-based operators such as Scale-invariant Feature Transforms, affine-invariant patches and Histogram of Oriented Gradients, Geometric Blur, as well as features inspired by neuroscience like the V1-based model (for a description of these and their reference see Section 1.1 in [1]). In order to expand the applicability of Machine Learning to feature extraction tasks, much recent work has focussed on making object recognition systems less dependent on human-based feature construction. The aim is to design systems so that good feature representations can be automatically learned to support effective induction of a classifier. The fundamental question that is addressed here is: given an image, how should we compute its representation?

Many recent object recognition systems use a cascade of two basic modules responsible for feature construction [1] followed by a classifier induction algorithm, generally a multinomial logistic regression. These modules are: (1) a 2D convolutional-based transformation of patches of raw pixel intensity values into a 2D *feature map*, and (2) a pooling operation (i.e. down-sampling) that combines spatially nearby values of the feature map, for example through a max or average operator. The simpler implementations employ a single layer of feature detection, however, systems based on multiple layers of feature detectors are currently setting the winning records in object recognition competitions [2].

With the advent of *deep learning* [3], schemes for stacking layers of feature detectors with the aim of building "deep" feature representations have been proposed. The dominant method for training deep feature construction systems is the so-called *greedy layer-wise training* [4]. In this method, a hierarchy of features is learned one layer at a time: learning a new transformation at layer l that is composed of a previously learned transformation at layer $l-1$. After the end of greedy layer-wise training, the resulting features of the last layer can be used as input to train a classifier in supervised mode. It is empirically demonstrated that layer-wise stacking of feature detectors often yields a representation that is more efficiently classified as opposed to its single-layer counterpart [5].

The layer-wise learning of features can be performed with either unsupervised, or semi-supervised or supervised learning [3]. *Greedy layer-wise supervised training* (GLST), which is also the topic of the present work, was first introduced in the context of deep feed-forward neural networks in [6]. In that work, after the first one-hidden-layer network is trained, its output layer was discarded and a new one-hidden-layer network is stacked on top of it. The resulting network is trained using gradient descent, leaving unchanged the weight vector of the previous hidden layer. This process is repeated for a number of different layers. The generalisation performance of the proposed system was superior to the system having the same number of hidden layers that were jointly trained. Another variant of GLST is presented in the work of [7], where the outputs of the previous layer were fed as extra inputs, in addition to the raw input, for the next layer. Moreover, a successful application of GLST to conversational speech transcription is reported in [8].

Genetic Programming (GP) has been successfully applied to object recognition tasks (see [9] for a surveying table in page 5). Most systems evolve classification programs that select and non-linearly combine features from a predefined bag of features that are hand-crafted. More importantly, all of the GP systems to-date use a single stage of classifier evolution. Finally, multi-class object recognition usually requires either the evolution of multiple one-vs-the-rest binary classifiers or, in case where a standalone classifier is used, some sort of effective classification strategy in order to map its output into different class labels. Multinomial logistic regression is a very successful multi-class classification algorithm that has not been combined with GP to-date.

The paper proposes a novel way to evolve with GP a hierarchical feature construction and classification system with feedforward processing. The layered architecture of the system stacks one or two feature construction stages, each of which consists of a layer that transforms a number of input image representations into a collection of *feature maps*, and a pooling layer that combines the values over local neighbourhoods of a feature map using an average operation. The final layer of the architecture is a regularised logistic regression classifier.

Alexandros Agapitos is with the Complex and Adaptive Systems Laboratory, School of Computer Science and Informatics, University College Dublin, Ireland. (email: alexandros.agapitos@ucd.ie).

To our knowledge, this is the first ever work that addresses the method of greedy layer-wise supervised training as a form of deep learning with GP. For an initial step towards deep GP systems for object recognition, we are addressing the following research questions:

1) Is GP able to evolve image-feature representations from low-level image data (i.e. pixel intensity values), which perform better in object recognition tasks than basic hand-designed features based on statistical moments?
2) What is the effect of greedy layer-wise supervised training? Is there any advantage of using a deep feature representation, that is a system architecture with two successive stages of feature construction, rather than an architecture with a single-stage?

The rest of the paper is organised as follows. Section II presents a brief overview of object recognition in GP. Section III describes the proposed method. Section IV outlines the experiment setup. Section V presents the experimental results, while Section VI concludes and proposes future work.

II. Object recognition with GP

Object recognition is an area that has received some attention from the GP community. The purpose of this section is twofold: it first discusses GP systems in terms of feature construction; it then briefly reviews that main program architectures that have been used for object recognition to-date.

A. Feature construction

We identify two major classes of studies: (1) studies that evolve symbolic expressions composed of predefined features; (2) studies where the evolved symbolic expressions are composed of raw pixel-based input.

The first category of studies use GP terminal sets that contain predefined domain-independent features and/or domain-specific features. This involves pre-processing image data with low-level feature extraction algorithms. The works of [9], [10], [11], [12], to name some of the most recent ones, fall in this category.

The second category concerns the evolution of object classifiers that use raw pixel intensity values as input. The majority of function sets use operations that are statistical moments parameterised with (a) position coordinates and (b) size of an image patch that is used as input to the operation. Evolution allows for these parameters to be optimised as classifier programs are getting fitter. The works of [13], [14] fall in this category. In addition, there have been systems that use terminal sets containing as many variables as the number of pixels in an image patch of fixed size; for a recent example see [15].

B. Object recognition system architectures

The dominant program architecture in evolving object classifiers with GP is a standalone expression-tree that is responsible for simultaneous feature construction and classification. Arithmetic and other operations are used to transform terminals representing extracted features or pixel intensity values to a real-valued output at the root of the expression-tree, which is further mapped to a classification label. Two of the most recent systems based on an architecture that allows for simultaneous feature construction and classification are reported in [13], [14]. Both systems successfully evolved standalone classifiers from raw pixel-based input using the classification accuracy as the fitness function.

The work of [10] was one of the first to propose an architecture in which a feature extraction program is independently evolved from the classification program. In order to first train the feature extraction stage, the human identifies regions of "feature" and "non-feature" using a graphical user interface. A program is then evolved to detect such features. The evolution of a classifier that uses the "pre-trained" feature detectors is based on the *boosting* methodology, in which partial solutions are trained on different distribution of training examples and are gradually added to the overall classifier.

A modular feed-forward architecture is reported in the work of [15]. It is defined by cascading a transformation layer, a pooling layer and a classification layer. A moving-window-based extraction of image patches is combined with an evolved transformation to convert an original image into a transformed image. The transformed image is then down-sampled using several statistical moments, and the resultant vectorised representation of the transformed image is used as input to a Nearest-Neighbour classifier or a Decision-tree classifier. The system is required to evolve an image representation from raw pixel-based input that is effectively classified using a very simple classification algorithm.

The vast majority of existing systems evolve standalone programs that simultaneously perform feature extraction and classification in a single evolutionary run. In addition, most problems tackled are binary classification problems that require the output of the program to be mapped to a class label by setting the value of zero as the discriminating threshold between classes. Evolving multi-class classifiers in often difficult with standalone programs; addressing the problem via the evolution of a collection of one-vs-the-rest classifiers often improves classification performance. How can we make GP classifiers scale-up to hundreds or even thousands of classes? See for example the ImageNet dataset with 1000 classes (http://www.image-net.org/) or the Caltech-101 dataset with 101 classes (http://www.vision.caltech.edu/Image_Datasets/Caltech101/).

III. Deep Evolution of Feature Representations

The architecture of the system, which is inspired by [1], is a stack of layers defined in a bottom-up fashion as follows:

1) Filter bank layer.
2) Transformation layer.
3) Average pooling (i.e. down-sampling) layer.
4) Classification layer.

A. Filter bank layer

A *filter bank* is a collection of filters, which are represented as 2D arrays of values (i.e. weights) that are used

as kernels in 2D discrete convolution operations [1]. There exist various choices for the filters composing the bank in this layer (for a description see Section 1.1 in [1]). These can either be predefined or randomly initialised and subsequently learned. In this initial implementation the values of the 2D arrays are *initialised to random values and they are kept fixed* (no evolution takes place on these parameters). The justification of this choice is twofold.

First and foremost, it has been shown [1], [16] that convolutional pooling architectures enable even random-value filters to perform competitively against systems where the filter banks are learned in an unsupervised or supervised way. The authors in [1], [16] conclude that while filter fine-tuning is essential to achieve state-of-the-art performance, the major contribution to a system's performance can be attributed to the choice of its architecture in terms of different ways for cascading modules of transformation and pooling.

Second, as an initial step, we are interested in studying only the evolution of the transformation layer. This is somewhat of a more controlled experiment with a more limited search space as opposed to a system that allows for the evolution of both the filter bank and transformation layers.

The filter bank is a collection of K filters (also known as *kernels*) $\{k_1, \ldots, k_K\}$, where filter $k_i \in \mathbb{R}^{w \times w}$ with w referred to as the *receptive field size*. The input to the filter bank layer is an image $Img \in \mathbb{R}^{n \times n}$, represented as a $n \times n$ matrix of pixel intensity values. The output is a 3D representation $(n-w+1) \times (n-w+1) \times K$ (with K filters). Each $(n-w+1) \times (n-w+1)$ *feature map* fm_i is generated using a 2D discrete convolution operation $(k_i * Img)$ between the filter k_i and the image. The convolution operation moves the $w \times w$ filter across the image with a step-size of one pixel. Figure 1 illustrates a sample generation of a feature map through the 2D discrete convolution operation [1]. In the following abbreviation, the superscript denotes the receptive field size of the filters, while the subscript denotes the number of filters in the filter bank. For example $F^{(5\times5)}_{(10)}$ denotes a filter bank layer with 10 filters of dimensionality 5×5. The output from this layer is 10 feature maps.

B. Transformation layer

The transformation layer receives an array of m feature maps, that is a $d \times d \times m$ representation, where $d \times d$ is the dimensionality of each feature map. The transformation process extracts patches from this 3D representation using a moving-window-based approach of step-size of one pixel. Each patch has dimension $w \times w \times m$ with w referred to as the *receptive field size* and m as the number of channels. A patch $p \in \mathbb{R}^{w \times w \times m}$ is transformed into $y \in \mathbb{R}$ via an evolved program $f : \mathbb{R}^{w \times w \times m} \mapsto \mathbb{R}$. The program output y is then passed through the hyperbolic tangent function before it is assigned as the intensity value of the destination pixel in the transformed feature map. By sliding a moving window with a step-size of one, the entire input representation of $d \times d \times m$ is transformed into a $(d-w+1) \times (d-w+1)$ representation. Figure 4 illustrates an example. The abbreviation $T^{(3\times3)}$ denotes, for example, a transformation layer with a receptive field size of 3×3.

[1]Figure taken from https://developer.apple.com/library/ios/documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html. Last accessed in 14/01/2015.

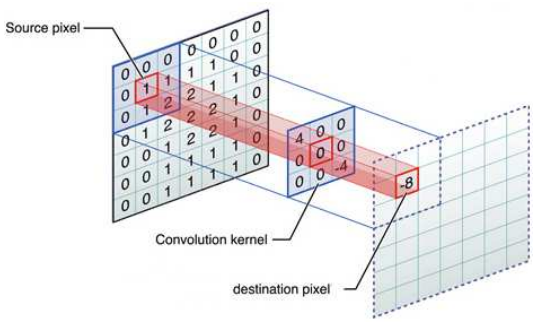

Fig. 1. Sample generation of a 5×5 feature map as the result of the 2D discrete convolution of a 7×7 input image with a 3×3 filter. The center element of the filter is placed over the source pixel. The value of the destination pixel in the feature map is then calculated as $(4 \times 0) + (0 \times 0) + (0 \times 0) + (0 \times 0) + (0 \times 1) + (0 \times 1) + (0 \times 0) + (0 \times 1) + (-4 \times 2)$. The complete feature map is generated by placing the center of filter over all possible source pixels in the input image with the step-size of one pixel. The surrounding border pixels shown in the feature map are ultimately discarded resulting in 5×5 representation.

C. Average pooling layer

This layer is used to reduce the dimensionality of the representation via down-sampling. An $m \times m$ feature map down-sampled with a $d \times d$ pooling window will result in a $(m/d) \times (m/d)$ feature map. Each output value is $y_{ijk} = \sum_{pq} w_{pq} \cdot x_{i,j+p,k+q}$, where w_{pq} is a uniform weighting window. An example of average pooling is illustrated in Figure 2. An average pooling layer with a 4×4 down-sampling is abbreviated to $P^{(4\times4)}$.

D. Classification layer

The classification layer receives a $s \times s \times m$ representation, where $s \times s$ is the dimensionality of a feature map and m the number of channels (i.e. feature maps). This layer accepts feature maps that were already processed by evolved programs. The system is able to use either a single evolved program for classification, in which case $m = 1$, or use multiple evolved programs, in which case $m > 1$. The vectorised representation of each feature map is first generated, that is a vector in $\mathbb{R}^N$, where $N = s \cdot s$. Given m such vectors, a total of $m \cdot N$ features are used for classification. We employ a regularised multinomial logistic regression classifier, which is trained with a cross-entropy loss function using a Quasi-Newton optimisation method. The ridge parameter λ is set to the value of 0.001 and the number of iterations to 1000. An example classification layer that aggregates features from 3 channels is abbreviated to $C^{(3)}$.

E. Greedy layer-wise supervised training

Different object recognition systems can be assembled by cascading the above-mentioned layers in different ways. For example, F – T – P – C is a basic feed-forward object recognition system that uses a filter bank layer, followed by a transformation layer, an average down-sampling layer and finally a classification layer.

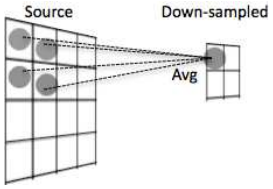

Fig. 2. Example of average pooling of a 4×4 feature map with a 2×2 pooling window. The pooling window is positioned on all possible non-overlapping areas of the feature map, computing the average of pixel intensity values from the extracted patches. The resultant representation is a 2×2 feature map.

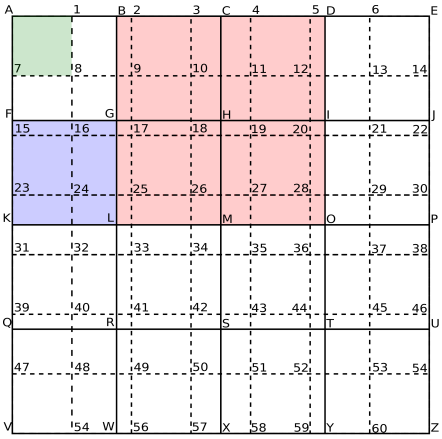

Fig. 3. Set of hand-made features. 74 feature areas where identified and the mean and std. deviation of each calculated, resulting in 148 features. The first set of features consist of subdividing the image into 16 non-overlapping squares of size 7x7 pixels. An example of this type of feature is highlighted above in the square F G L K. The second set of features contains 9 overlapping squares 14x14 pixels in size, seen above in the highlighted square B D O L. Finally the image is divided into 49 non-overlapping squares of size 4x4 pixels. This type of feature is highlighted in the square A 1 8 7 above.

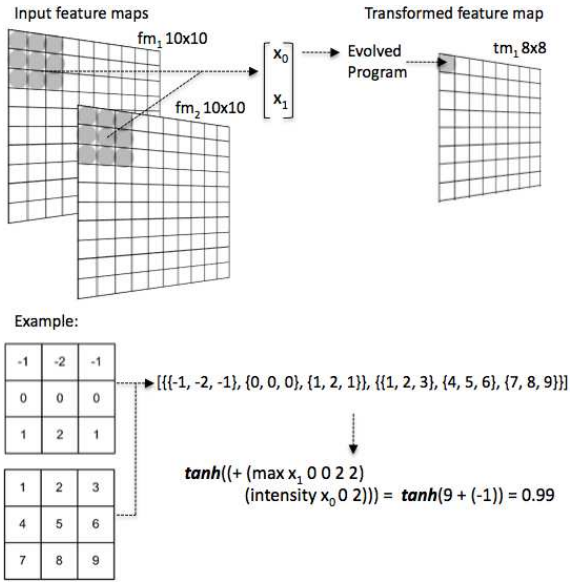

Fig. 4. Example of a $T^{(3\times3)}$ transformation layer that transforms a $10 \times 10 \times 2$ representation into a 8×8 representation. The input to the evolved program is a 3D array of pixel intensity values representing the two extracted 3×3 patches. The sample program adds the maximum value from the second matrix to the value in position (0, 2) from the first matrix. See later section for description of primitive elements used by evolved programs.

Fig. 5. Sample images from MNIST dataset.

This section describes the training protocol considered for learning deep feature representations with GP. The algorithm, which is based on [6], is *supervised*, *greedy*, and *layer-wise*. The general approach is to train each new layer of image transformation in an independent training run, taking each time as input the output of the last of previously trained transformations. Every successive transformation layer serves as pre-training initialisation of a newly-invoked training procedure with the aim to learn a stack of gradually better representations.

During the first stage of evolution, we evolve the first transformation layer T_{StageA} using the architecture of F – T_{StageA} – P – $C^{(1)}$. A placeholder in T_{StageA} allows to test different programs in terms of their ability to generate effective image representations (i.e. feature maps). The *wrapper* approach to feature map evolution is taken, in which the logistic regression classifier C is wrapped around the feature map produced by a candidate program, with the *classification error rate* accruing from the classifier assigned as the fitness of the candidate feature map. At the end of the evolutionary run a number K of best-evolved feature maps (i.e. maps that were generated from K best programs) of dimensionality $d \times d$ are aggregated in a $K \times d \times d$ representation abbreviated to $T_{StageA,K}$.

In the second stage of evolution, a second transformation layer T_{StageB} is evolved using as input the output of the first evolutionary run, that is a $K \times d \times d$ representation of best-evolved feature maps. A new, independent evolutionary run is invoked using the system architecture $T_{StageA,K}$ – T_{StageB} – P – $C^{(1)}$. Note that the layer F used in the first stage is now replaced by $T_{StageA,K}$. The rest of the procedure for evolving T_{StageB} remains the same as above.

IV. Experiment Setup

A. MNIST dataset

We tackle a very popular Machine Learning benchmark (10-class classification), the MNIST benchmark, which contains 28×28 grey-level images of hand-written digits. The dataset contains $60,000$ training examples and $10,000$ test examples, and it is publicly available from http://yann.lecun.com/exdb/mnist/. In the current work we use the complete test set, but only train on the first $30,000$ training examples. This was chosen in order to reduce the memory requirements of our initial system implementation. A sample of training images is given in Figure 5.

B. Systems under comparison

We are experimenting with four different systems for evolving hand-written digit classifiers, specified as follows:

1) **Single-stage transformation system** (SST). Its architecture is defined as $F_{50}^{(5\times5)} - T^{(5\times5)} - P^{(4\times4)} - C^{(1)}$. The first layer is composed of a filter bank of 50 filters with 5×5 receptive fields, resulting in a $24 \times 24 \times 50$ representation. 10 filters were generated from each of $U(-1.0, 1.0)$, $U(-5.0, 5.0)$, $UD(1, 5)$, $N(1.0)$, $N(5.0)$, where $U(x, y)$ denotes uniform sampling of real-valued numbers within the $[x, y]$ interval, $UD(x, y)$ denotes uniform sampling of discrete numbers in $[x, y]$ interval, and $N(x)$ denotes sampling from a Normal distribution with a standard deviation set to x. The same filters are used in all independent evolutionary runs. The transformation layer uses a 5×5 receptive field, which further reduces the resolution of the representation to 20×20. The down-sampling layer uses an average pooling of 4×4, which produces a final output feature map of size 5×5. Based on a single program (i.e. $C^{(1)}$), this generates 25 features that are then fed to the logistic regression classifier that produces a vector of 10 outputs representing the probability distribution over class labels (i.e. $\{0, \ldots, 9\}$). The *classification error rate* is used as fitness function for evolving the transformation layer. This is defined as $1.0 - \frac{correct}{N}$, where $correct$ is the number of examples that are correctly classified and N is the total number of examples in the training set.

2) **Double-stage transformation system** (DST). The second stage feature construction system is fed with the output of the first stage, denoted as $T_{StageA,50}$. The subscript 50 denotes that the second stage uses the best 50 evolved feature maps from the first stage. The overall architecture is defined as $T_{StageA,50} - T^{(5\times5)} - P^{(4\times4)} - C^{(1)}$. The transformation layer uses a 5×5 receptive field size operating on 20×20 feature maps, which further reduces the resolution of the representation to 16×16. The down-sampling layer uses an average pooling of 4×4, which produces a final output feature map of size 4×4, totalling 16 distinct features for logistic regression classification based on a single program. The classification error rate is used as the fitness function.

3) A standard GP system that evolves programs with real-valued output (STGP), which is mapped to a class label via a *classification map* [11]. This method positions class regions sequentially on the floating point line. An input is classified to the class of the region that program-output falls into. We defined the following map using the identical interval of 1.0 between class labels: $0 \in (-\infty, 1.0)$, $1 \in [1.0, 2.0)$, $2 \in [2.0, 3.0)$, $3 \in [3.0, 4.0)$, $4 \in [4.0, 5.0)$, $5 \in [5.0, 6.0)$, $6 \in [6.0, 7.0)$, $7 \in [7.0, 8.0)$, $8 \in [8.0, 9.0)$, $9 \in [9, +\infty)$. Using this program representation we experiment with three different fitness functions: **(i)** $STGP_{mse}$. The first fitness function takes the form of mean squared error between program output p_i and desired output y_i for the i^{th} training example. Given a set of N examples, this is formally defined as $\frac{1}{N}\sum_{i=1}^{N}(p_i - y_i)^2$, where $y_i = classlabel_i + 0.5$ with $classlabel \in \{0, \ldots, 9\}$. **(ii)** $STGP_{mse/ls}$. The second fitness function is based on the first one, but performs a linear bias correction of p_i (known as linear scaling in the GP literature) prior to calculating its squared deviation from y_i. The slope b and intercept a are calculated so as to minimise the squared error between y_i and $a + bp_i$. That is, $b = \frac{1}{N}\sum_{i=1}^{N}[(p_i - p')(y_i - y')] / \sum_{i=1}^{N}(p_i - p')^2$, where p', y' denote the average program output and average target value respectively. The means squared error is defined as $\frac{1}{N}\sum_{i=1}^{N}(a + bp_i - y_i)^2$. **(iii)** $STGP_{er}$. The third fitness function takes the program output and directly maps it to a class label using the classification map specified above. The classification error rate is used as fitness function.

4) A GP system that evolves programs taking the form of *decision-trees* (GPDT). A decision-tree is a hierarchical structure composed of nested `if-then-else` constructs, each taking 3 arguments. The first argument is a predicate expression that is evaluated to either true or false. The second and third arguments are either class labels or `if-then-else` constructs. If the result of the predicate's evaluation is `true` then the second argument is returned; else, the third argument is returned. The fitness function used for evolving decision-trees is the classification error rate described above.

C. GP systems setup

Tables I and II summarise the functions and terminals respectively. The arithmetic functions of division, logarithm and square root are protected. Both SST and DST systems use the same functions and terminal elements. The function set contains arithmetic and image-based functions. `Min`, `Max`, `Mean`, `StdDeviation`, `Entropy`, each receives five arguments. The first argument represents an `ImagePatch`. The second, third, fourth, and fifth arguments represent $X_{UpperLeft}$, $Y_{UpperLeft}$, $X_{LowerRight}$, $Y_{LowerRight}$ `Coordinate` values respectively. These functions return the respective statistical operation applied to the *rectangle* in the `ImagePatch` that the four parameters specify. The smaller of the two X values is interpreted as $X_{UpperLeft}$ and the larger is interpreted as $X_{LowerRight}$. The same is done for the Y values, thus the four parameters always specify a legal rectangle in the `ImagePatch`. The `Coordinate` terminals are not allowed to go beyond the `ImagePatch` boundaries, so given receptive fields of size 5×5 used in the transformation layers, `Coordinate` values are generated within the range $\{0, 4\}$. Finally, the `Intensity` function returns the intensity value of a pixel in an `ImagePatch`. Its second and third arguments specify X, Y `Coordinate` values respectively.

The STGP system uses arithmetic functions, while GPDT uses both arithmetic and decision-tree functions. 148 `ImageFeature` terminals are used in STGP and GPDT systems. These are the means and std. deviations of certain regions in the 28×28 example images. The way in which the regions are defined is illustrated in Figure 3. In addition, `RandDouble` terminals are used by STGP, while GPDT uses both `RandDouble` and `ClassLabel` terminals.

SST uses a population size of 500 evolved for 100 generations. DST takes as input the 50 best feature maps evolved by generation 50 of SST and evolves them for an extra 50 generations, thus the computational effort of the two systems is the same. Both systems use tournament selection, with a tournament size of 4. On the other hand, STGP

TABLE I. STRONGLY-TYPED FUNCTIONS

Function	Argument(s) type	Return type
Arithmetic		
$+, -, *, /$	double, double	double
e^x, log(x), sqrt(x), sin(x), tanh(x)	double	double
Image-based		
Min, Max, Mean, StdDev, Entropy	image, integer, integer integer, integer	double
Intensity	image, integer, integer	double
Decision-tree		
If-Then-Else	boolean, integer, integer	integer
and, or, xor	boolean, boolean	boolean
not	boolean	boolean
$<, \leq, >, \geq$	double, double	double

TABLE II. STRONGLY-TYPED TERMINALS

Terminal	Type	Description
RandDouble	double	randomly generated double in [-1.0, 1.0] interval
ImageFeature	double	means and std. deviations of certain regions (Figure 3)
ImagePatch	image	2D array $height \times width$ of pixel intensity values
Coordinate	integer	$x \in [0, width-1]$ or $y \in [0, height-1]$
ClassLabel	integer	class label in $\{0, \ldots, 9\}$

and GPDT systems use $1,000$ individuals evolved for 100 generations. The population size was deliberately set higher after obtaining very poor preliminary runs with a population of 500 individuals. Tournament size is set to 7.

All four systems were initialised with the ramped-half-and-half method with depths between 2 and 6. The maximum allowed depth during evolution was set to 12. Subtree crossover (90% inner nodes, 10% leaf nodes), subtree mutation (max. depth of random tree set to 4), and point-mutation (prob. of a node to be mutated set to $1.0/TreeSize$) were applied with probabilities of 30%, 40% and 30% respectively. Elitism of 1% of population size was used.

D. Dynamic sampling of training examples

We used a variant of *Dynamic Subset Selection*, which is a hybrid between *Random-per-Generation* (RPG) and *Random-per-Individual* (RPI) methods. A random sample R_g of size $3,000$ for generation g is made up of two samples RPG_g and RPI_{gj}. RPG_g is a random sample that is used by all members of the population, while RPI_{gj} is a random sample independently drawn for individual j.

The algorithm for RPG involves randomly selecting a target number $S = 2,500$ of examples from the complete training set of $N = 30,000$ examples per generation, with a bias, so that an example is more likely to be selected based on its *difficulty*. Each training example i is assigned a weight $w_i = (1.0 + \frac{m}{n})^d$, where m is the number of individuals misclassified the example, n is the number of individuals that tackled the example at a particular generation, and d is the exponent of the polynomial weighting scheme set to the value of 9 in our experiments. At the first generation both m and n are set to the value of zero. The probability that a training example i will be selected at generation g is given by $P_{ig} = \frac{w_{ig} \cdot S}{\sum_{z=1}^{N} w_{zg}}$. We iterate through the training set $\forall i : 1 \leq i \leq N$, picking at each iteration a random number r in the range of [0.0, 1.0], and selecting example i if $P_{ig} > r$. The selected sample size fluctuates around the target size S, while the inclusion of S in the calculation of P_{ig} ensures that the expected selected subset size is of target size. If an example is selected to be included in the subset, the values of m and n are initialised to zero.

Once the first random sample of size T has been generated using the RPG method, a number of $T' = 3,000 - T$ examples are randomly picked independently for each individual. RPI uses a uniform sampling among $30,000$ training examples. The process ensures that the RPG and RPI samples form disjoint sets.

V. RESULTS

We performed 30 independent evolutionary runs for each system on Section IV-B. Training is based on dynamic samples drawn from the complete set of $30,000$ examples as described above. Generalisation performance in assessed via the classification error rate on the complete test set of $10,000$ examples that is provided in the MNIST distribution. For the sake of brevity we will refer to it simply as error rate in the remaining of this section.

Figures 6(a),(b)(c) show the evolution of error rate for $STGP_{mse}$, $STGP_{mse/ls}$, and $STGP_{er}$ respectively. Results suggest that $STGP_{er}$ outperforms the rest of the systems, attaining a median error rate of 0.63 at generation 100 as opposed to the median values of 0.78 for $STGP_{mse/ls}$ and 0.82 for $STGP_{mse}$. We can conclude that the fitness function based on the training classification error rate is the most appropriate when evolving classifiers in the form real-valued-output expression-trees, which require a classification map for mapping program-outputs to class labels. Furthermore, linear scaling (Figure 6(b)) offers improvements over MSE alone (Figure 6(a)).

In the case of GPDT (Figure 6(d)), the median error rate of 0.48 by generation 100 outperforms all $STGP$ systems. This means that 52% of test cases are correctly classified. It seems that the decision-tree representation results in a more *evolvable* system for the case of this multi-category classification task. In fact, there is a lack of studies in the literature that deal with many classes, and the issue of classifier representation and scalability to such problems surely warrants further investigation. A final observation that is consistent in all four standard GP systems is that there is no overfitting; the error curves take the form of monotonically decreasing functions of the generation number. It is therefore interesting to study longer evolutionary runs in the future.

Figure 6(e) shows the evolution of error rate for the case of SST using a single best-evolved individual to generate features in the classification layer. Median error rate reaches a minimum of 0.076 by generation 50, at which point evolution seems to stagnate and slight overfitting is evident thereafter. Figure 6(f) shows the evolution of error rate for the case of DST, using the best-evolved feature vector at the final classification layer. By generation 30, a median error rate of 0.070 is obtained, while by generation 40 the median error rate is 0.069. Overfitting is observed past generation 40.

We performed a *Wilcoxon rank sum test* to test the *null* hypothesis that error rate data at SST/Gen50 and error rate

data at DST/Gen30 are samples from continuous distributions with equal medians, against the *alternative* that they are not. The obtained *p*-value is 0.0011 with DST outperforming SST. An additional Wilcoxon rank sum test between error rate data at SST/Gen50 vs DST/Gen40 obtained a *p*-value of 0.0007 with DST outperforming SST.

Having evolved a population of programs generating the transformed feature maps, we were tempted to use more than the single best-evolved program to generate features in the classification layer. We decided to use the 5 best-evolved or 10 best-evolved programs from each population. In the case of 5 programs, this led to $25 \cdot 5 = 125$ features in SST, and in $16 \cdot 5 = 80$ features in DST. Additionally, in the case of 10 programs, this led to $25 \cdot 10 = 250$ features in SST, and in $16 \cdot 10 = 160$ features in DST.

The results for 5 best programs are illustrated in Figure 6(g). Contrasting between Figures 6(e) and 6(g), we observe that the error rate is significantly improved from a median of 0.076 to the value of 0.046 in SST/Gen50. In the case of DST/Gen30 the error rate is improved from a median value of 0.070 to the value of 0.045. However the differences in median values between SST/Gen50 and DST/Gen30 are not statistically significant in Figure 6. Why multi-program features didn't benefit from two stages of transformation? The observation that only the single best-program evolution significantly benefited from DST can be attributed to the fact that the fitness function was rewarding the evolution of classifiers that were exclusively using either 25 features in SST or 16 features in DST. We believe that the evolution of sets of multiple programs can benefit from a two stage transformation only in case where the system is deliberately evolved towards that goal; this means that fitness is based on the performance of a logistic regression classifier that explicitly use a bigger number of features.

In summary, using 5 best-evolved programs can be beneficial to the generalisation performance as opposed to a single best program. But using 10 best programs (Figure 6 worsens generalisation compared to 5 best programs. The aggregation of several features extracted from different best-evolved programs is currently naive, given that we explicitly evolved towards classifiers with either 25 or 16 features. This is main reason of overfitting that was observed in 10 best programs as opposed to 5 best programs. Nevertheless, why did 5 best programs performed better that the single best program? It seems that they benefited from the dynamic sampling of a portion of training examples independent for each individual, which allowed for some sort of semantic diversity to be maintained in the population. The logistic regression classifier was able to leverage on these diverse features, and a more powerful model was obtained.

Finally, Figure 6(i) shows the error rate of SST/Gen50 vs. the error rate of DST/Gen30 in 30 independent evolutionary runs for different number of best-evolved programs used for classification. In the case of single best-evolved programs, two layers of transformation is critical in improving performance. In the case of 5 or 10 best programs, the results are mixed.

VI. Conclusion and Future Work

This paper was motivated by the latest research in developing efficient learning algorithms for deep architectures of feature construction systems, since these has been proven to be much more representationally expressive than shallow ones. In this work we demonstrated the successful application of greedy layer-wise supervised training of a deep, feed-forward GP-based system for handwritten digit recognition. The proposed system, operating on raw pixel-based input, outperformed several standard GP system setups with hand-designed features. Results also revealed that multi-category classification is a very hard problem for standard GP that uses a standalone expression-tree to handle all different classes.

We are barely scratching the surface of this exciting niche of research. Some future research questions are as follows:

- What is the impact of different forms of pooling, i.e. max-pooling layer? (see Section 2 in [1]).
- How can we evolve towards cooperative sets of programs in order to involve more than a single best-evolved program for classification? In this context, how can we prevent complex co-adaptations of the programs in a population (that may lead to overfitting) so as to evolve independent feature detectors?
- The literature suggests that convolution is an architectural feature that plays a significant part in providing good classification performance, and was used as the first layer of the object recognition system. How can we adapt the filter bank during evolution?
- We are currently focussing on the method of stacking single-layer image transformations. How can we address the joint, simultaneous evolution of all the layers?

References

[1] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun, "What is the best multi-stage architecture for object recognition?" in *Proc. International Conference on Computer Vision (ICCV'09)*. IEEE, 2009.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*.

[3] Y. Bengio, A. C. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1798–1828, 2013.

[4] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, Jul. 2006.

[5] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin, "Exploring strategies for training deep neural networks," *J. Mach. Learn. Res.*, vol. 10, pp. 1–40, Jun. 2009.

[6] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Advances in Neural Information Processing Systems 19*, B. Schölkopf, J. Platt, and T. Hoffman, Eds. Cambridge, MA: MIT Press, 2007, pp. 153–160.

[7] D. Yu, S. Wang, and L. Deng, "Sequential labeling using deep-structured conditional random fields." *J. Sel. Topics Signal Processing*, vol. 4, no. 6, pp. 965–973, 2010.

[8] F. Seide, G. Li, X. Chen, and D. Yu, "Feature engineering in context-dependent deep neural networks for conversational speech transcription." in *ASRU*, D. Nahamoo and M. Picheny, Eds. IEEE, 2011, pp. 24–29.

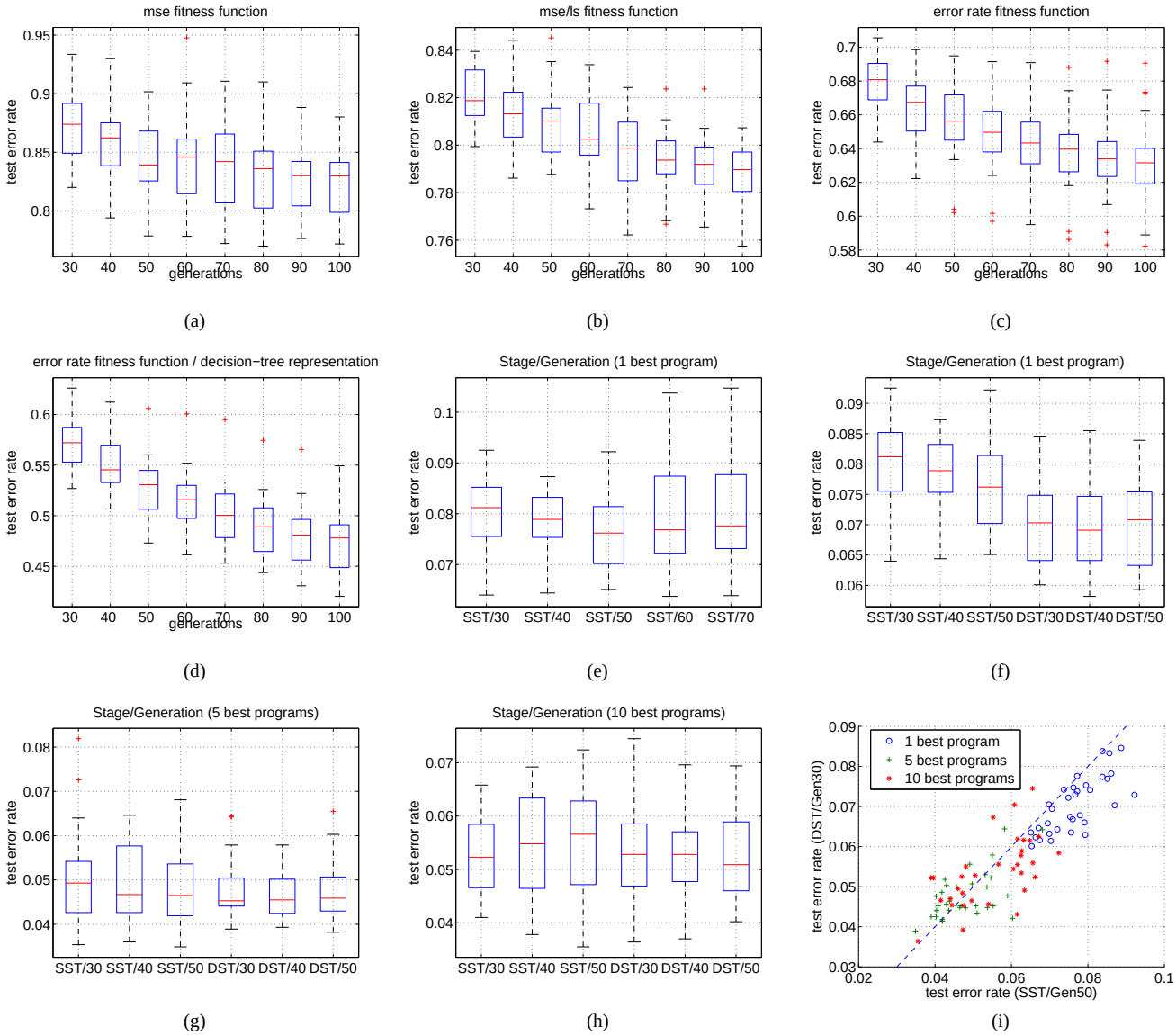

Fig. 6. Box-plots depicting distributions of test classification error rates are generated based on 30 independent evolutionary runs. Figures (a), (b), (c) show the evolution of error rates for $STGP_{mse}$, $STGP_{mse/ls}$, $STGP_{er}$ respectively. Figure (d) shows the evolution of error rate for GPDT. Figures (e), (f) show the evolution of error rate for SST and DST, using a single best-evolved program in the classification layer. Figure (g), (h) shows the performance of DST when accumulating features from 5 best-evolved programs and 10 best-evolved programs respectively. Figure (i) shows error rate in generation 50 of SST vs. error rate in generation 30 of DST (scatter plot of 30 independent runs). The line y=x is also illustrated to facilitate comparison of error rates.

[9] M. Zhang, X. Gao, and W. Lou, "A new crossover operator in genetic programming for object classification," *IEEE Transactions on Systems, Man and Cybernetics, Part B*, vol. 37, no. 5, pp. 1332–1343, Oct. 2007.

[10] O. Oechsle and A. F. Clark, "Feature extraction and classification by genetic programming," in *6th International Conference Computer Vision Systems, ICVS 2008*, ser. Lecture Notes in Computer Science, A. Gasteratos, M. Vincze, and J. K. Tsotsos, Eds., vol. 5008. Springer, 2008, pp. 131–140.

[11] M. Zhang and W. Smart, "Multiclass object classification using genetic programming," in *Applications of Evolutionary Computing, EvoWorkshops2004*, ser. LNCS, vol. 3005. Springer Verlag, 2004.

[12] Y. Li, J. Ma, and Q. Zhao, "Two improvements in genetic programming for image classification," in *2008 IEEE WCCI*, J. Wang, Ed., IEEE Computational Intelligence Society. Hong Kong: IEEE Press, 1-6 Jun. 2008, pp. 2492–2497.

[13] H. Al-Sahaf, A. Song, K. Neshatian, and M. Zhang, "Two-tier genetic programming: towards raw pixel-based image classification," *Expert Systems with Applications*, vol. 39, no. 16, pp. 12 291–12 301, 2012.

[14] D. Atkins, K. Neshatian, and M. Zhang, "A domain independent genetic programming approach to automatic feature extraction for image classification," in *Proceedings of the 2011 IEEE CEC*, A. E. Smith, Ed., IEEE Computational Intelligence Society. New Orleans, USA: IEEE Press, 5-8 Jun. 2011, pp. 238–245.

[15] T. Kowaliw, W. Banzhaf, N. Kharma, and S. Harding, "Evolving novel image features using genetic programming-based image transforms," in *2009 IEEE Congress on Evolutionary Computation*, A. Tyrrell, Ed., IEEE Computational Intelligence Society. Trondheim, Norway: IEEE Press, 18-21 May 2009, pp. 2502–2507.

[16] A. Saxe, P. W. Koh, Z. Chen, M. Bhand, B. Suresh, and A. Ng, "On random weights and unsupervised feature learning," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ser. ICML '11, L. Getoor and T. Scheffer, Eds., 2011.