Incremental Evolution and Development of Deep Artificial Neural Networks

Filipe Assunção[1,2](✉), Nuno Lourenço[1], Bernardete Ribeiro[1], and Penousal Machado[1]

[1] CISUC, Department of Informatics Engineering,
University of Coimbra, Coimbra, Portugal
{fga,naml,bribeiro,machado}@dei.uc.pt

[2] LASIGE, Department of Informatics, Faculdade de Ciencias,
Universidade de Lisboa, Lisbon, Portugal

Abstract. NeuroEvolution (NE) methods are known for applying Evolutionary Computation to the optimisation of Artificial Neural Networks (ANNs). Despite aiding non-expert users to design and train ANNs, the vast majority of NE approaches disregard the knowledge that is gathered when solving other tasks, i.e., evolution starts from scratch for each problem, ultimately delaying the evolutionary process. To overcome this drawback, we extend Fast Deep Evolutionary Network Structured Representation (Fast-DENSER) to incremental development. We hypothesise that by transferring the knowledge gained from previous tasks we can attain superior results and speedup evolution. The results show that the average performance of the models generated by incremental development is statistically superior to the non-incremental average performance. In case the number of evaluations performed by incremental development is smaller than the performed by non-incremental development the attained results are similar in performance, which indicates that incremental development speeds up evolution. Lastly, the models generated using incremental development generalise better, and thus, without further evolution, report a superior performance on unseen problems.

Keywords: Incremental development · NeuroEvolution · Convolutional Neural Networks

1 Introduction

Automated Machine Learning (AutoML) is a sub-field of Artificial Intelligence (AI) that automates with little or no human-intervention the application of Machine Learning (ML) approaches to the user's problem, avoiding the need for the manual tuning of the data pre-processing, the design and extraction of features, and/or the selection and parameterisation of the most suitable ML model. The current work focuses on a branch of AutoML: NeuroEvolution (NE) [1]. NE applies Evolutionary Computation (EC) to search for Artificial Neural Networks

T. Hu et al. (Eds.): EuroGP 2020, LNCS 12101, pp. 35–51, 2020.
https://doi.org/10.1007/978-3-030-44094-7_3

(ANNs), enabling the optimisation of their structure (e.g., number of neurons, layers, connectivity), and/or learning (i.e., weights, or learning algorithm and its parameters). In other words, the ultimate goal of NE is to empower non-expert ML users with the ability to design effective ANNs.

One of the main limitations of NE lies in the fact that the majority of the methods only address a specific problem, i.e., the ANNs are evolved for one task, and when there is the need to solve a new problem the entire search procedure is re-started from scratch. Therefore, the methods do not take advantage of any of the information available from addressing previous similar tasks. In addition, NE approaches tend to evolve large populations of individuals that are continuously optimised throughout a usually large number of generations. The evaluation of a single ANN is time-consuming, because it often requires the training of the networks with a defined (or evolved) learning strategy. Consequently, the search for effective ANNs resorting to NE tends to be slow. This problem is even more striking when optimising Deep Artificial Neural Networks (DANNs).

In this work we extend Fast Deep Evolutionary Network Structured Representation (Fast-DENSER) [2] to incremental development, i.e., we transfer and re-use the knowledge acquired when optimising DANNs (architectures and learning strategies) to previous problems, and cumulatively apply it to learn new classification tasks. The main contributions of this work are the following:

- The extension of the Fast-DENSER framework to incremental development;
- The demonstration that DANNs evolved by incremental development statistically outperform the canonical approach;
- The indication that incremental development speeds up evolution. When given the same number of generations, incremental development surpasses the performance of the evolution from scratch. For the same level of performance fewer generations are necessary;
- The evidence that the method works as expected in terms of evolution, i.e., knowledge from previously solved problems is introduced in any stage;
- The conclusion that the DANNs that are evolved by incremental development generalise better than those obtained by the non-incremental version. The performance of the incrementally generated DANNs is superior to their independent evolution counterparts in previously addressed, and in yet unaddressed problems.

The remainder of the document is organised as follows. Section 2 surveys related works in the field of NE applied to DANNs, and incremental development; Sect. 3 details Fast-DENSER; Sect. 4 introduces the extension of Fast-DENSER to incremental development; Sect. 5 presents the experimental setup and results; and Sect. 6 draws conclusions and addresses future work.

2 Related Work

NeuroEvolution (NE) approaches are usually grouped according to the target of evolution, i.e., topology [3,4], learning (i.e., weights, parameters, or learning policies) [5–7], or the simultaneous evolution of the topology and learning [8,9]. Nonetheless, more recent efforts have been put towards the proposal of methods that deal with the optimisation DANNs, and thus we feel that it is more intuitive to divide them into small-scale [5,8] and large-scale [7,10–13] NE. The current paper focuses on the latter; a complete survey can be found in [14].

The problem of most of the methods that target the evolution of DANNs is that, even aided by Graphics Processing Units (GPUs) they tend to take a lot of time to find effective models. For example, CoDeepNEAT [10] trains on 100 GPUs, and Real et al. use 450 GPUs for 7 days to perform each run [15]. Fast-DENSER takes approximately 4.7 days with a single GPU to perform each run, and that is the reason why we have selected Fast-DENSER for the current paper. There are methods that are computationally cheaper, e.g., Lorenzo and Nalepa [16] take about 120 min to obtain results; however, the speedup is obtained at the cost of the performance of the model.

To speedup evolution some authors have investigated the use of transfer learning in NE. The main goal of transfer learning is to make use of the knowledge acquired when solving previous tasks to facilitate the resolution of others, enhancing lifelong learning [17]. One of the most recurrent ideas is that of using past knowledge to provide a better start than random seeding (e.g., [18,19]).

A key problem on transfer (and even multi-task) learning is the representation. Verbancsics and Stanley [20] demonstrate that transfer learning is most effective when the representation is the same for the multiple problems that are to be addressed. That is one of the advantages of using a grammar-based NE approach such as Fast-DENSER: the grammar nature of the method makes passing from one task to the next one transparent, and requires no changes to the individuals' representation.

Whilst some transfer learning works seek to learn high-level features that are generalisable across multiple domains (e.g., [21]), our objective is to port individuals to warm start evolution to another problem, and in theory help to reach high performing solutions in less time. An example of a similar work, but where a hand designed network is used is introduced by Ciresan et al. [22], where there is the transfer of knowledge from Latin digits recognition to uppercase letters, and from Chinese characters to uppercase Latin letters.

3 Fast-DENSER

Fast-DENSER [2] is an extension of Deep Evolutionary Network Structured Representation (DENSER) [13]: a general-purpose grammar-based NE approach for optimising DANNs. DENSER can search for any type of DANN, and the target of evolution is specified in a Context-Free Grammar (CFG). An example of a CFG for encoding Convolutional Neural Networks (CNNs) is provided in Fig. 1. The typical structure of CNNs divides the topology into two parts: (i) layers for feature extraction (convolutional and pooling, lines 1–3), and layers for classification (fully-connected, line 11). The grammar of Fig. 1 explores these layer types, and also regularisation layers (dropout and batch normalisation, lines

```
         <features> ::= <convolution> | <convolution>                                          (1)
                        | <pooling> | <pooling>                                                 (2)
                        | <dropout> | <batch-norm>                                              (3)
      <convolution> ::= layer:conv [num-filters,int,1,32,256] [filter-shape,int,1,2,5]         (4)
                        [stride,int,1,1,3] <padding> <activation> <bias>                        (5)
       <batch-norm> ::=layer:batch-norm                                                         (6)
          <pooling> ::= <pool-type> [kernel-size,int,1,2,5]                                     (7)
                        [stride,int,1,1,3] <padding>                                            (8)
        <pool-type> ::= layer:pool-avg | layer:pool-max                                         (9)
          <padding> ::= padding:same | padding:valid                                           (10)
   <classification> ::= <fully-connected> | <dropout>                                          (11)
  <fully-connected> ::= layer:fc <activation>                                                  (12)
                        [num-units,int,1,128,2048 <bias>                                       (13)
          <dropout> ::=layer:dropput [rate,float,1,0,0.7]                                      (14)
       <activation> ::= act:linear | act:relu | act:sigmoid                                    (15)
             <bias> ::= bias:True | bias:False                                                 (16)
          <softmax> ::= layer:fc act:softmax num-units:10 bias:True                            (17)
         <learning> ::= <bp> <early-stop> [batch_size,int,1,50,500]                            (18)
                        | <rmsprop> <early-stop> [batch_size,int,1,50,500]                     (19)
                        | <adam> <early-stop> [batch_size,int,1,50,500]                        (20)
               <bp> ::= learning:gradient-descent [lr,float,1,0.0001,0.1]                      (21)
                        [momentum,float,1,0.68,0.99]                                           (22)
                        [decay,float,1,0.000001,0.001] <nesterov>                              (23)
         <nesterov> ::= nesterov:True | nesterov:False                                         (24)
             <adam> ::= learning:adam [lr,float,1,0.0001,0.1] [beta1,float,1,0.5,1]            (25)
                        [beta2,float,1,0.5,1] [decay,float,1,0.000001,0.001]                   (26)
          <rmsprop> ::= learning:rmsprop [lr,float,1,0.0001,0.1]                               (27)
                        [rho,float,1,0.5,1] [decay,float,1,0.000001,0.001]                     (28)
       <early-stop> ::= [early_stop,int,1,5,20]                                                (29)
```

Fig. 1. CFG for the optimisation of the topology and learning strategy of CNNs.

1–3, and 11). Furthermore, the grammar enables the optimisation of the learning strategy (learning, lines 18–20). The parameters of each evolutionary unit (in the current work layers or learning algorithms) are kept in the grammar, and can be integer (e.g., the filter shape in line 4), float (e.g., the momentum in line 22) or closed choice (e.g., the bias in line 16). The integer and float parameters are represented by a block with the format: [variable-name, variable-type, num-values, min-value, max-value].

In addition to the CFG we need to define the macro-structure, that establishes the search space, and points directly to the grammar production rules. The macro-structure sets the sequence of evolutionary units that the individuals are allowed to use, and is encoded as a list of tuples, where each position indicates the non-terminal symbol (that establishes a one-to-one mapping to the grammar, and is used as starting symbol), and the minimum and maximum number of expansions for that non-terminal symbol. For example, for CNNs, an example of a macro-structure is [(features, 1, 10), (classification, 1, 2), (softmax, 1, 1), (learning, 1, 1)]. This macro-structure allows for CNNs with between 3 and 13 layers, and where the learning strategy is optimised.

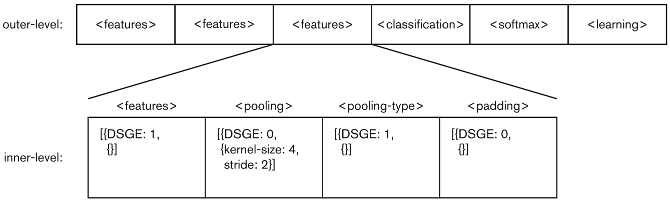

Fig. 2. Example of the genotype of a candidate solution that encodes a CNN.

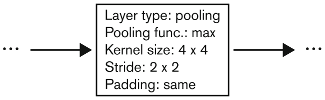

Fig. 3. Phenotype of the layer specified by the inner-level of Fig. 2

The genotype of the candidate solutions is organised into two levels: (i) the outer-level encodes the sequence of evolutionary units (with respect to the macro-structure), and sets the non-terminal symbol that is used as initial symbol for the grammatical derivation; and (ii) the inner-level corresponds to each outer-level position and encodes the parameters of a specific evolutionary unit. The inner-level genotype is similar to the genotype of Dynamic Structured Grammatical Evolution (DSGE); for more details on DSGE refer to [23]. An example of the genotype and corresponding phenotype of a candidate solution are represented in Figs. 2 and 3, respectively.

The representation of the candidate solutions in DENSER and Fast-DENSER is the same. The differences between the two approaches lie in the evolution of the population and in the evaluation of the candidate solutions. In DENSER evolution is conducted as in a standard Genetic Algorithm, where in each generation a large population of individuals is evaluated and offspring is generated. Contrary, Fast-DENSER follows a (1+λ) Evolutionary Strategy (ES), and therefore in each generation fewer individuals are evaluated. The results have demonstrated that Fast-DENSER, with the same individual evaluation scheme, can generate individuals that have the same quality as those generated by DENSER, in a fraction of the time. More precisely, there is a speedup of 20x from DENSER to Fast-DENSER. In addition, Fast-DENSER is extended to enable the generation of fully-trained DANNs, i.e., networks that need no further training by the end of the evolutionary process. To this end, Fast-DENSER evaluates the individuals for a maximum GPU training time. However, the maximum training

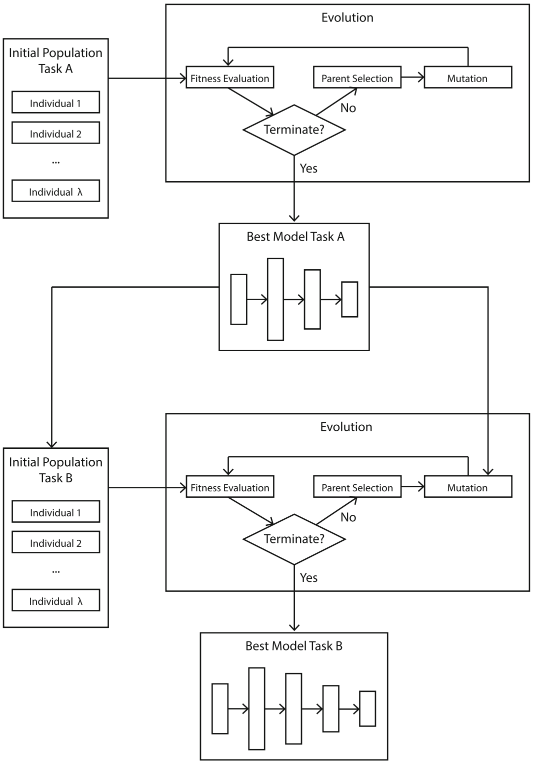

Fig. 4. Incremental development Fast-DENSER flow-chart.

time granted to each individual can grow continuously as required. The networks that are likely to benefit from longer training cycles are given access to a greater evaluation time as evolution proceeds.

4 Incremental Development of Deep Neural Networks

Experiments on previous work have shown that Fast-DENSER, given the same computational time budget, can obtain results that are superior to those reported by DENSER. The results are achieved without taking advantage of any of the knowledge acquired when solving other problems. In this paper we investigate the impact of building the networks incrementally, i.e., we take into account the DANNs that are generated for solving previous related problems, speeding up evolution, and possibly finding more effective solutions.

The flow-chart that illustrates the extension of Fast-DENSER to incremental development is depicted in Fig. 4. It shows how the method proceeds to address two different tasks A and B. For the first problem, task A, the method works similarly to Fast-DENSER: an initial population is randomly created and evolves until the stop criterion is met. The difference occurs when we solve a new problem, given that we have information on a prior one. For task B, the creation of the initial population takes into account the best model found for a previous problem (in this case task A). During evolution the past knowledge can also be incorporated. This rationale is generalised for more than two problems, i.e., in case we later address a task C, we use the knowledge obtained when addressing tasks A and B. Next, we will discuss how the prior knowledge is introduced in the initial population, and during evolution.

The initial population is formed by individuals that can be either entirely generated at random or that can use sets of evolutionary units from past models. The evolutionary units are transferred taking into account the macro-structure. For example, considering the macro-structure introduced above for CNNs, [(features, 1, 10), (classification, 1, 2), (softmax, 1, 1), (learning, 1, 1)], the initial population can contain individuals that (i) have all the layers comprising the feature extraction, and generate the classification layers at random; (ii) generate at random the feature extraction layers, and copy the layers that perform classification from previous models; (iii) copy only the learning evolutionary unit, and generate all the remaining ones at random; (iv) generate all evolutionary units at random, not using any previous knowledge; or (v) any other possible combination. It is important to mention that this incremental development approach only focuses on the evolutionary units, and consequently the weights are not transferred from previous models. At most we allow the learning strategy (which is an evolutionary unit) to be ported.

The models generated for solving each of the previously addressed problems are also important during evolution. The mutations in Fast-DENSER are tailored for manipulating DANNs: they enable the addition, removal, and/or duplication of any evolutionary unit, and the perturbation of the integer and/or float values. The duplication mutation, as the name suggests, replicates a given evolutionary unit by reference, and thus, any mutation that later affects this evolutionary unit changes all of its copies. In the incremental development version of Fast-DENSER the duplication can copy evolutionary units either from the individual or from any of the best models that were generated for solving previous tasks.

The individuals are evaluated only on the new problem. Therefore, up to the moment, this method is incremental in the sense that the DANNs for solving new and unseen problems do not kick off evolution from scratch. The incremental development does not mean that by the end of evolution the generated models can solve multiple tasks. However, it is expected that the models that are built considering previous knowledge generalise better than those that are always evolved from a random population. That is, we expect the models generated by incremental development to perform well in other tasks when re-trained.

5 Experimentation

To compare the incremental and non-incremental versions of Fast-DENSER we consider four computer vision datasets: MNIST, SVHN, Fashion-MNIST, and

Table 1. Description of the datasets.

Dataset	Train set size	Test set size	Number of classes	Shape
MNIST	60000	10000	10	$28 \times 28 \times 1$
SVHN	73257	26032	10	$32 \times 32 \times 3$
Fashion-MNIST	60000	10000	10	$28 \times 28 \times 1$
CIFAR-10	50000	10000	10	$32 \times 32 \times 3$

CIFAR-10 (summarised in Sect. 5.1). In particular, we conduct experiments for the following setups: (i) MNIST; (ii) SVHN; (iii) Fashion-MNIST; (iv) CIFAR-10; (v) MNIST → SVHN; (vi) MNIST → SVHN → Fashion-MNIST; (vii) MNIST → SVHN → CIFAR-10. The symbol → denotes the incremental build of the model from one task to the next. The setups are chosen according to the relatedness and expected difficulty of the tasks: the MNIST and SVHN datasets are composed by digits, and then transferred to two different domains, Fashion-MNIST, and CIFAR-10. The parameters required for the conducted experiments are detailed in Sect. 5.2. The experimental results are divided into three sections. First, in Sect. 5.3 we analyse the evolutionary performance when evolving DANNs for MNIST, SVHN, Fashion-MNIST, and CIFAR-10 with and without incremental development. Second, in Sect. 5.4 we investigate the incremental development of the topologies. Third, in Sect. 5.5, we analyse the generalisation ability of the different models. The experimental results are discussed in Sect. 5.6.

5.1 Datasets

The experiments are conducted in 4 datasets: MNIST, SVHN, Fashion-MNIST, and CIFAR-10. The characteristics of the datasets are summarised in Table 1. The shape of the instances is formatted as width × height × number of channels; grayscale images have one channel, and RGB images have three channels. A brief overview of the dataset instances is provided next.

MNIST [24] – handwritten digits from 0 to 9. The instances are pre-processed: size-normalized, and centered;

SVHN [25] – digits gathered from real-world images from house numbers in Google Street View images;

Fashion-MNIST [26] – similar to MNIST, where the images of handwritten digits are replaced by fashion clothing items: top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot;

CIFAR-10 [27] – real-world pictures of objects that are of one of the following classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

5.2 Experimental Setup

The experimental parameters are detailed in Table 2. The table is organised into 4 sections: (i) evolutionary engine – parameters related to Fast-DENSER (1+λ)-ES; (ii) dataset – parameters concerned with the dataset partitioning; (iii) training – parameters associated with the training of the DANNs; and (iv) data augmentation – parameters required for the dataset augmentation strategy.

The number of generations is different for each of the datasets. In particular, we perform 20, 30, 50, and 100 generations for the MNIST, Fashion-MNIST, SVHN, and CIFAR-10 datasets, respectively. The number of generations for each dataset was set empirically based on previous experiments, and according to how challenging each problem is expected to be. The grammatical mutation rate is a DSGE parameter that stands for the probability of changing any of the grammar expansion possibilities or integer/float parameter values.

The dataset section of Table 2 defines how we partition the train set of each dataset, i.e., for each run the train set is divided (in a stratified way) into three independent folds: (i) evolutionary train – used for training the DANNs; (ii) evolutionary validation – used to perform early stop; and (iii) evolutionary test – used to measure the fitness of the individual, which is measured using the accuracy. The test set (that is different from the evolutionary test set) is kept

Table 2. Experimental parameters.

Evolutionary engine parameter	Value
Number of runs	10
Number of generations	20/30/50/100
λ	5
Add layer rate	25%
Duplicate layer rate	15%
Remove layer rate	25%
Grammatical mutation rate	15%
Dataset parameter	**Value**
Evolutionary Validation set	3500 instances
Evolutionary Test set	3500 instances
Evolutionary Train set	Remaining instances
Test set	Check Table 1
Training parameter	**Value**
Training Time	10 min
Loss	Categorical Cross-entropy
Data augmentation parameter	**Value**
Padding	4
Random crop	4
Horizontal flipping	50%

out of evolution and is used only after the end of the evolutionary search. It measures how well the models behave beyond the data used during evolution, and enables the unbiased evaluation of the performance.

The datasets, as discussed in Sect. 5.1, have different shapes: MNIST and Fashion-MNIST are $28 \times 28 {\times} 1$, and SVHN and CIFAR-10 are $32 \times 32 \times 3$. To facilitate the application of the optimised DANNs to all datasets we reshape the MNIST and Fashion-MNIST to $32 \times 32 \times 3$. The image width and height are resized using the nearest neighbour method, and to pass from one to three channels we replicate the single channel three times. All the datasets are applied the same data augmentation strategy: padding, random cropping, horizontal flipping, and re-scaling to $[0, 1]$. We do not subtract the mean image nor normalize.

Table 3. Average performance of the optimised DANNs. The results are averages of 10 independent runs. Bold marks the highest average performance values.

Dataset	Evolutionary accuracy	Test accuracy
MNIST	98.86 ± 0.465	98.80 ± 0.298
SVHN	93.28 ± 0.863	93.31 ± 0.955
MNIST → SVHN	**94.01 ± 0.891**	**94.04 ± 0.887**
Fashion	92.42 ± 1.224	91.41 ± 1.049
MNIST → SVHN → Fashion	**93.92 ± 0.930**	**92.96 ± 0.742**
CIFAR-10	87.18 ± 1.242	86.19 ± 1.672
MNIST → SVHN → CIFAR-10	**89.06 ± 1.488**	**88.19 ± 1.669**

The networks are trained for an initial maximum GPU time of 10 min, and thus it is important to mention that we are performing each evolutionary run in a GeForce GTX 1080 Ti GPU. For the experiments conducted on this paper we use the grammar of Fig. 1, and the macro-structure: [(features, 1, 30), (classification, 1, 10), (softmax, 1, 1), (learning, 1, 1)]. The code for Fast-DENSER can be found in the GitHub repository https://github.com/fillassuncao/fast-denser3.

5.3 Experimental Results: Incremental Development

We start by comparing the DANNs generated by Fast-DENSER with and without incremental development in terms of performance. The results are summarised in Table 3. We report the evolutionary accuracy (i.e., fitness), and the test accuracy (i.e., the accuracy of the models on an unseen partition of the datasets). The results are averages of 10 independent runs. The first conclusion is that given the same computational time (number of generations), the results reported by the incremental development are always superior to those of when evolution starts from scratch. In particular, the performance of MNIST → SVHN is superior to the performance of SVHN, the performance of MNIST → SVHN → Fashion is superior to the performance of Fashion, and the performance of MNIST → SVHN → CIFAR-10 is superior to the performance of CIFAR-10.

To better acknowledge the differences between the fittest DANNs generated with and without incremental development we use statistical tests. To check if the samples follow a Normal Distribution we use the Kolmogorov-Smirnov and Shapiro-Wilk tests, with $\alpha = 0.05$. The tests reveal that the data does not follow any distribution and thus the non-parametric Mann-Whitney U test ($\alpha = 0.05$) is used to perform the comparisons between the setups. The statistical tests show that the results of MNIST → SVHN → Fashion, and MNIST → SVHN → CIFAR-10 are statistically superior (in evolution and test) to Fashion (evolutionary p-value = 00736, test p-value = 0.00278), and CIFAR-10 (evolutionary p-value = 0.00804, test p-value = 0.01552), respectively. The effect size is large for all the statistically significant comparisons ($r > 0.5$). The difference between MNIST → SVHN and SVHN is not statistically significant (evolutionary p-value = 0.05876, test p-value = 0.0536). With only 20 generations the MNIST setup is the one that attains the highest average accuracy results. This indicates that it is an easy to solve problem and consequently no much knowledge is acquired from addressing it. This is a well-known fact: a simple fully-connected network is able to attain good performances in the MNIST dataset.

The above results prove that incremental development, given the same number of generations, designs DANNs that outperform those generated without incremental development. On the other hand, what happens when, for each setup, we only let evolution to be conducted for a smaller amount of generations, so that the cumulative number of generations is not superior to that of when evolution is conducted from scratch? The cumulative number of generations is the sum of the number of generations of each incremental step. For example, for the MNIST → SVHN, the cumulative number of generations is 70 $(20 + 50)$. In this scenario we consider 30, 0, and 30 generations for the MNIST → SVHN, MNIST → SVHN → Fashion, and MNIST → SVHN → CIFAR-10 setups, respectively. The average evolutionary performance of the 10 fittest networks slightly decreases to 93.69 ± 0.912, 92.91 ± 1.15, and 87.13 ± 2.225, respectively for the MNIST → SVHN, MNIST → SVHN → Fashion, and MNIST → SVHN → CIFAR-10 setups. With these results there is no statistical difference for any of the setups, i.e., with incremental development, given a cumulative search time that equals the search time from scratch, we are able to generate DANNs that report the same performance as those optimised without incremental development for more generations. In other words, the use of previous knowledge speeds up evolution.

In addition to analysing the average performance over the 10 evolutionary runs we also focus on the overall best found DANN, i.e., the fittest DANN among the conducted runs. This analysis is important considering that in a real-world scenario by the end of evolution what really interests the user is the best found model, which is the one to potentially be deployed live. To avoid an unbiased choice of the best model for each dataset, the decision is taken only with regard to the evolutionary performance. The results are reported in Table 4, and once again show that the best results are obtained by incremental development. The most striking result is the one of CIFAR-10, where the difference introduced by incremental development is the highest.

Table 4. Accuracy of the best performing DANN for each of the setups. Bold marks the highest performance value.

Dataset	Evolutionary accuracy	Test accuracy
MNIST	99.46	99.12
SVHN	94.20	93.88
MNIST → SVHN	**94.80**	**94.14**
Fashion	93.91	92.92
MNIST → SVHN → Fashion	**94.80**	**93.92**
CIFAR-10	88.74	88.14
MNIST → SVHN → CIFAR-10	**91.06**	**89.79**

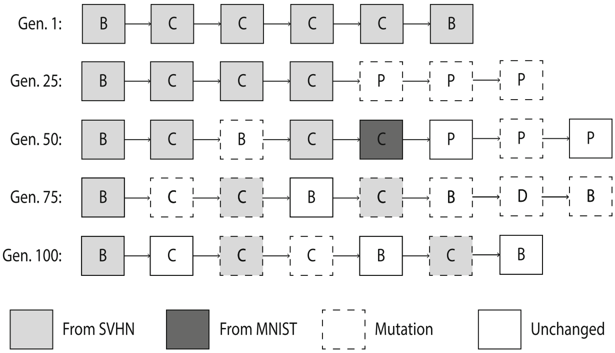

Fig. 5. Overview of the evolution on the incremental development setup MNIST → SVHN → CIFAR-10. We provide a snapshot of the feature-layers of the best individual on the 1st, 25th, 50th, 75th, and 100th generations. For space constraints we focus on the feature extraction layers: Convolutional (C), Pooling (P), Batch-Normalization (B), and Dropout (D).

5.4 Experimental Results: Topology Analysis

To analyse the behaviour of incremental development from a structural point of view we inspect the topology of the best networks as evolution proceeds. Figure 5 shows the evolution of the structure of the networks on the setup MNIST → SVHN → CIFAR-10. Because of space constraints we select the setup where more generations were performed, and present the snapshots of the run that generates the DANN with the median fitness value, i.e., we order the runs according to the fitness of the best generated DANN and select the 6th run. We choose the median run to avoid a biased selection over the worst or best results. We focus only on the feature extraction layers. The figure's goal is to illustrate the exploration of knowledge incorporation, and thus the parameters of the layers are omitted.

The figure makes it evident that the amount of layers that come from previously addressed tasks without any change diminishes as evolution proceeds. That is the expected behaviour: in the initial generation the fittest DANN reuses all layers from the best network generated to address the SVHN, and across generations these layers are adapted to tackle the CIFAR-10 (e.g., convolutional in generation 75). During evolution new layers are also randomly created (e.g., batch-normalization in generation 50), and others removed (e.g., dropout in generation 100). Similarly to the the non-incremental approach, new random layers can be added, but in addition, in the incremental development strategy we can also add layers that come from the previously solved tasks (e.g., convolutional layer that is transferred from the MNIST in generation 50).

The snapshots prove that incremental development is able to generate better results based on the re-use of evolutionary units that aid solving previous problems. The evolutionary units are not only incorporated in the generation of the initial population, but also during evolution. We also inspect the evolutionary results of other setups and the conclusions are inline with the reported.

5.5 Experimental Results: Generalisation of the Models

With the objective of studying the generalisation ability of the generated models we measure their performance on all the considered datasets. For example, we take the best generated solutions for the MNIST dataset and apply them to the SVHN, Fashion and CIFAR-10 datasets without further evolutionary optimisation. The networks are re-trained on the target datasets with the same topology and learning strategy that is optimised for the source task. Table 5 summarises the test results for all the setups. The values in bold mark the best generalisation performance, i.e., the best performance of the setup (row) that has not yet seen the dataset (column), e.g., for the CIFAR-10 dataset (last column), except for the setups that specifically target this dataset (CIFAR-10, and MNIST→SVHN→CIFAR-10), the setup that attains the highest performance is MNIST→SVHN→Fashion, and thus this is the setup that is marked in bold.

The analysis of the results shows that incremental development always generates better results, even for tasks that have not been addressed previously. To better understand the differences we perform a statistical analysis, and compare the performances reported by the non-incremental and incremental approaches. Therefore we compare the SVHN and MNIST→SVHN setups on the MNIST, Fashion, and CIFAR-10 datasets, and we do similarly with the remaining pairs: Fashion vs. MNIST→SVHN→Fashion, and CIFAR-10 vs. MNIST→SVHN→CIFAR-10. The same conditions of the above statistical comparison are applied. The statistical tests reveal that there are only significant differences between the Fashion, and MNIST→SVHN→Fashion setups, with p-values of 0.02574, and 0.01732, respectively for the SVHN and CIFAR-10 datasets (the effect size is large). The direct comparison for the dataset used for evolution (in this case Fashion) was performed above and revealed a statistical significance in favour of incremental development for the setups that include two incremental development steps.

Table 5. Performance of the evolved DANNs when applied to other datasets. The results are averages of 10 independent runs, where each DANN is trained 5 times. The setups are the table rows, and the datasets the columns.

	MNIST	SVHN	Fashion	CIFAR-10
MNIST	98.80 ± 0.298	71.31 ± 29.60	90.17 ± 1.842	63.63 ± 23.29
SVHN	96.87 ± 5.426	93.31 ± 0.955	91.60 ± 1.289	78.49 ± 7.899
MNIST→SVHN	98.93 ± 0.266	94.04 ± 0.887	91.83 ± 1.312	82.58 ± 2.414
Fashion	92.73 ± 16.75	89.16 ± 3.551	91.41 ± 1.049	77.32 ± 4.893
MNIST→SVHN→Fashion	98.89 ± 0.273	$\mathbf{92.48 \pm 2.167}$	92.96 ± 0.742	$\mathbf{83.47 \pm 2.294}$
CIFAR-10	99.06 ± 0.039	90.18 ± 9.282	92.91 ± 0.479	86.19 ± 1.672
MNIST→SVHN→CIFAR-10	$\mathbf{99.11 \pm 0.071}$	90.08 ± 5.924	$\mathbf{93.16 \pm 0.3328}$	88.19 ± 1.669

In case we order the datasets by difficulty, given by the non-incremental test performance on each dataset, we have MNIST, SVHN, Fashion, and CIFAR-10, where the leftmost is the easiest one, and the rightmost is the most challenging. From these results we hypothesise that superior generalisation performances are obtained by incremental development, when passing from more simple to more challenging datasets. That is the reason why there is no statistical difference in the CIFAR-10 vs. MNIST→SVHN→CIFAR-10 setups: the CIFAR-10 is per-se more challenging to solve than the remaining ones, and therefore, as already noticed in a previous article [13], the DANNs generated for addressing CIFAR-10 tend to be able to solve other easier problems. The remarkable aspect of incremental development is when a DANN optimised for Fashion is able to get better results on the CIFAR-10, compared to when the DANNs for Fashion are not evolved in an incremental fashion.

5.6 Discussion

The results presented in the previous sections compare in terms of performance, topology, and generalisation ability the search conducted by non-incremental and incremental development. The evolutionary results show that given the same search time the DANNs obtained by incremental development statistically outperform the non-incremental counterparts. On the other hand, the incremental strategy speeds up evolution, and given the same cumulative search time reports results that match the non-incremental performances.

The speedup in evolution is facilitated by the warm-start of incremental development, and possibility to still incorporate knowledge from previous tasks by mutation as generations proceed. We show an example of this by representing several snapshots of a network across generations. In particular for the selected run of the MNIST→SVHN→CIFAR-10 setup, on the first generation the best individual replicates all the layers from the MNIST→SVHN setup, which are continuously modified and adapted to the CIFAR-10. During evolution the parameters of the layers that are copied from the previous setup are changed, new layers (random, and from previous setups) are added, and others removed. That is, the behaviour of the incremental development evolution is the expected.

Finally, we analyse the generalisation ability of the generated DANNs. Without further evolution, i.e., with the same topology and learning strategy obtained when optimising a DANN for a specific task, we re-train the DANNs on the remaining datasets. The results show that, on average, the incremental development results are superior to the non-incremental results. Moreover, the results are statistically significant when the generated DANNs are applied to a more difficult task than that where they were generated. This indicates that incremental development helps in learning increasingly more challenging tasks, and that there are not major differences when performing the opposite.

6 Conclusions and Future Work

Motivated by the difficulty and burden in the design of DANNs we investigate how to incorporate past knowledge to aid evolution. In particular, we extend Fast-DENSER – a general-purpose grammar-based NE framework – to take advantage of the evolutionary units acquired when optimising DANNs for previous tasks. This novel incremental developmental approach enables the incorporation of knowledge from any of the previously addressed tasks in any stage of evolution: both during the generation of the initial population, and by the application of mutations, as the generations proceed.

The results prove that incremental development improves the search performed by Fast-DENSER enabling it to obtain statistically superior results. Additionally, incremental development speeds up evolution, being able to obtain the same results as non-incremental evolution given the same cumulative search time, i.e., less generations are used for the target dataset. In addition, the DANNs obtained by the end of evolution generalise better when we use incremental development in the search: the networks designed for easy problems perform better in more challenging and yet unseen tasks.

The future work will target three different directions: (i) apply the incremental development methodology to a wider set of tasks and domains; (ii) extend the approach to modular evolution; and (iii) seek ways to transfer not only the evolutionary unit but also the weights (in case the evolutionary units are layers).

Acknowledgments. This work is partially funded by: Fundação para a Ciência e Tecnologia (FCT), Portugal, under the PhD grant agreement SFRH/BD/114865/2016, the project grant DSAIPA/DS/0022/2018 (GADgET), and is based upon work from COST Action CA15140: ImAppNIO, supported by COST (European Cooperation in Science and Technology): www.cost.eu.

References

1. Floreano, D., Dürr, P., Mattiussi, C.: Neuroevolution: from architectures to learning. Evol. Intell. **1**(1), 47–62 (2008)
2. Assunção, F., Lourenço, N., Machado, P., Ribeiro, B.: Fast DENSER: efficient deep neuroevolution. In: Sekanina, L., Hu, T., Lourenço, N., Richter, H., García-Sánchez, P. (eds.) EuroGP 2019. LNCS, vol. 11451, pp. 197–212. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16670-0_13

3. Gruau, F., Whitley, D., Pyeatt, L.: A comparison between cellular encoding and direct encoding for genetic neural networks. In: Proceedings of the 1st Annual Conference on Genetic Programming, pp. 81–89. MIT Press, Cambridge (1996)
4. Miller, G.F., Todd, P.M., Hegde, S.U.: Designing neural networks using genetic algorithms. In: ICGA, pp. 379–384. Morgan Kaufmann (1989)
5. Whitley, D.: Applying genetic algorithms to neural network learning. In: Proceedings of the Seventh Conference (AISB89) on Artificial Intelligence and Simulation of Behaviour, pp. 137–144. Morgan Kaufmann Publishers Inc. (1989)
6. Gomez, F.J., Schmidhuber, J., Miikkulainen, R.: Accelerated neural evolution through cooperatively coevolved synapses. J. Mach. Learn. Res. **9**, 937–965 (2008)
7. Stanley, K.O., D'Ambrosio, D.B., Gauci, J.: A hypercube-based encoding for evolving large-scale neural networks. Artif. Life **15**(2), 185–212 (2009)
8. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evol. Comput. **10**(2), 99–127 (2002)
9. Turner, A.J., Miller, J.F.: Cartesian genetic programming encoded artificial neural networks: a comparison using three benchmarks. In: GECCO, pp. 1005–1012. ACM (2013)
10. Miikkulainen, R., et al.: Evolving deep neural networks. CoRR abs/1703.00548 (2017)
11. Suganuma, M., Shirakawa, S., Nagao, T.: A genetic programming approach to designing convolutional neural network architectures. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 497–504. ACM (2017)
12. Real, E., et al.: Large-scale evolution of image classifiers. In: ICML. Proceedings of Machine Learning Research, vol. 70, pp. 2902–2911. PMLR (2017)
13. Assunção, F., Lourenço, N., Machado, P., Ribeiro, B.: Denser: deep evolutionary network structured representation. Genet. Program. Evolvable Mach. **20**, 5–35 (2018). https://doi.org/10.1007/s10710-018-9339-y
14. Baldominos, A., Saez, Y., Isasi, P.: On the automated, evolutionary design of neural networks: past, present, and future. Neural Comput. Appl. **32**(2), 519–545 (2019). https://doi.org/10.1007/s00521-019-04160-6
15. Real, E., Aggarwal, A., Huang, Y., Le, Q.V.: Regularized evolution for image classifier architecture search. arXiv preprint arXiv:1802.01548 (2018)
16. Lorenzo, P.R., Nalepa, J.: Memetic evolution of deep neural networks. In: GECCO, pp. 505–512. ACM (2018)
17. Thrun, S.: Is learning the n-th thing any easier than learning the first? In: NIPS, pp. 640–646. MIT Press (1995)
18. Tirumala, S.S., Ali, S., Ramesh, C.P.: Evolving deep neural networks: a new prospect. In: ICNC-FSKD, pp. 69–74. IEEE (2016)
19. Wong, C., Houlsby, N., Lu, Y., Gesmundo, A.: Transfer learning with neural AutoML. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31, pp. 8356–8365. Curran Associates, Inc. (2018)
20. Verbancsics, P., Stanley, K.O.: Evolving static representations for task transfer. J. Mach. Learn. Res. **11**, 1737–1769 (2010)
21. Long, M., Cao, Y., Wang, J., Jordan, M.I.: Learning transferable features with deep adaptation networks. arXiv preprint arXiv:1502.02791 (2015)
22. Ciresan, D.C., Meier, U., Schmidhuber, J.: Transfer learning for Latin and Chinese characters with deep neural networks. In: IJCNN, pp. 1–6. IEEE (2012)

23. Lourenço, N., Assunção, F., Pereira, F.B., Costa, E., Machado, P.: Structured grammatical evolution: a dynamic approach. In: Ryan, C., O'Neill, M., Collins, J.J. (eds.) Handbook of Grammatical Evolution, pp. 137–161. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78717-6_6
24. Lecun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
25. Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., Ng, A.Y.: Reading digits in natural images with unsupervised feature learning. In: NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011 (2011)
26. Xiao, H., Rasul, K., Vollgraf, R.: Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms (2017)
27. Krizhevsky, A., Hinton, G., et al.: Learning multiple layers of features from tiny images. Technical report, Citeseer (2009)