Epigenetic evolution of deep convolutional models

Alexander Hadjiivanov
School of Computer Science and Engineering
University of New South Wales
Sydney, Australia
a.hadjiivanov@student.unsw.edu.au

Alan Blair
School of Computer Science and Engineering
University of New South Wales
Sydney, Australia
blair@cse.unsw.edu.au

***Abstract*—In this paper, we build upon a previously proposed neuroevolution framework to evolve deep convolutional models. Specifically, the genome encoding and the crossover operator are extended to make them applicable to layered networks. We also propose a convolutional layer layout which allows kernels of different shapes and sizes to coexist within *the same* layer, and present an argument as to why this may be beneficial. The proposed layout enables the size of individual kernels within a layer to be evolved with a corresponding new mutation operator. The framework employs a hybrid optimisation strategy involving structural changes through epigenetic evolution and weight update through backpropagation in a population-based setting. Experiments on several image classification benchmarks demonstrate that the crossover operator is sufficiently robust to produce increasingly performant offspring even when the parents are trained on only a small random subset of the training dataset in each epoch, thus providing direct confirmation that learned features and behaviour can be successfully transferred from parent networks to offspring in the next generation.**

I. Introduction

Neuroevolution (NE), or the process of evolving the structure and/or the weights of neural networks (NNs), has matured into a viable and versatile optimisation tool over the past three decades. Evolution tends to converge slowly and generally requires a large number of evaluations, so early work on NE was limited to relatively small networks [1], [2]. As evolutionary algorithms grew in sophistication and the power and availability of hardware improved, NE was able to achieve excellent results in tasks of varying complexity [3], with a number of incremental improvements in genome encoding and evolution efficiency (e.g., Symbiotic Adaptive NeuroEvolution (SANE) [2], Enforced Sub-Populations (ESP) [4], Evolution Strategy with Covariance Matrix Adaptation (CMA-ES) [5], NeuroEvolution of Augmenting Topologies (NEAT) [6] and Cooperative Synapse NeuroEvolution (CoSyNE) [7]) seen around the turn of the century [8]. The success of NEAT gave rise to variants such as Hypercube-based NEAT (HyperNEAT), which uses NEAT to evolve a Compositional Pattern-Producing Network (CPPN) as a compact indirect encoding of the actual phenotype [9], [10]. Interestingly, CPPNs can evolve not only the structure and weights but also the transfer function of the NN nodes, which is a somewhat rare mutation operation in NE. More recently, Cartesian Genetic Programming (CGP) has been applied to directed graphs (rather than tree-based structures, which are commonly used in GP) to evolve both feed-forward and recurrent NNs [11] as well as heterogeneous networks with evolved transfer functions (similar to CPPNs) [12], which have achieved excellent results on dynamic control and classification tasks.

Within the last few years, research on NE has expanded from tasks which could be solved by evolved networks with relatively few weights, such as pole balancing [13] and robot maze navigation in a synthetic environment [14], to more complex visual tasks such as image classification [15]. Importantly, recognising the advantages of evolution as a global optimiser, there has been a paradigm shift towards utilising NE as an optimiser for the network structure in combination with backpropagation (BP) to fine-tune the network weights. For instance, deep convolutional NNs (CNNs) with multiple layers and millions of parameters have been evolved for tasks ranging from image classification [16], [17], image captioning [17] (using an evolved deep Long Short-Term Memory (LSTM) network) and even applications in particle physics (neutron scattering model selection) [18]. A differentiable version of CPPN was proposed in [19] to efficiently compress the representation of deep CNNs. Furthermore, genetic algorithms [20], particle swarm optimisation (PSO) [21] and GP [22] have also demonstrated excellent results on searching for optimal CNN structures for image classification tasks. In [23], an additional degree of complexity was explored by allowing local and long-range recurrent feedback connections to be discovered by evolution. As an extreme example of parallel NE, in [24] a massively parallel distributed environment was set up on top of the BOINC[1] platform to evolve highly optimised convolutional networks that achieve state-of-the-art performance on the MNIST dataset with a reported accuracy of 99.43%.

Evolution is commonly used to optimise the network architecture and minimise the number of parameters without compromising performance. For example, evolved CNN topologies have achieved the highest accuracy on an image classification task (Fashion-MNIST) compared to ten other popular models (including AlexNet and GoogLeNet) with

The first author is grateful for the Research Training Program (RTP) scholarship provided by the Australian Government.

[1]The Berkeley Open Infrastructure for Network Computing is a generic distributed computing platform.

a small fraction of the weights of the largest compared model (GoogLeNet) [25]. Similar results (up to 12-fold reduction in the number of parameters without loss of accuracy) were achieved through iterative connection pruning and retraining of large pretrained models (such as VGG-16 and AlexNet) on image classification tasks [26]. Another interesting approach proposed recently is MetaQNN [27], where state-of-the-art performance on image classification tasks is achieved with simpler network architectures discovered through reinforcement learning (RL). Although not using evolution, the latter two examples clearly demonstrate the merit of architecture search and further strengthen the case for exploring new avenues for research on NE. Despite the fact that evolution is usually used for optimising the structure rather than the weights, it was recently demonstrated [28] that evolution strategies can successfully optimise the weights of a NN with millions of parameters, achieving results rivalling those obtained with RL on a number of OpenAI Gym 3D tasks and Atari games while offering a number of advantages such as higher performance under distributed training and lower sensitivity to the temporal scale of the simulation.

II. Background

In previous research [29], we proposed a NE framework (named Cortex) which is based on a direct genotype encoding using the ordered number of nodes in an unstructured network as a straightforward metric for matching network topologies during crossover. In this paper, this framework is extended to make it applicable to deep layered NNs, with particular focus on deep CNNs. This section provides a brief overview of a typical CNN architecture and the Cortex NE framework (particularly genome encoding, speciation and crossover), which are extended in several ways in III to make them applicable to deep CNN evolution.

A. Convolutional networks

Convolutional networks have enjoyed an exponential surge in popularity since they have demonstrated extremely high performance in various domains of machine learning (particularly classification tasks relying on pattern recognition), such as document recognition [30] and image classification [31], [32]. The canonical convolutional network architecture consists of one or more feature extraction layers followed by one or more classification (fully connected; FC) layers (Fig. 1).

Convolutional layers are characterised by their kernel size K, stride S and padding P (Fig. 2), where the kernel size is regarded as a property of the layer itself. In this study, the extent of the kernel in the width and height dimensions is taken as a property of each individual kernel, which enables kernels of different sizes and shapes to be evolved.

B. Cortex NE framework

In Cortex, a distinction is made between a genome (a 'skeleton' of ordered nodes shared by all networks with the same number of nodes; Fig. 3a), a genotype (a genome with added connections but without weight information; Fig. 3b) and a phenotype (a genotype with weights assigned to all connections; Fig. 3c). The genome is used for *speciation*, which is a form of niching introduced together with NEAT [33] as way to group similar network genomes. Speciation was designed to improve the chance of survival of networks after structural mutations (such as adding or removing nodes or connections), which are likely to reduce the network's fitness initially, even though they might be beneficial in the long run. Using this type of speciation, individuals compete only with other individuals in the same species rather than with the entire ecosystem[2], which increases the chance of survival of unfit individuals.

The genotypes defined by a particular genome (species) are used for crossover, which is convenient because the nodes in the genotype are *ordered*, meaning that all genotypes can be aligned automatically. Although the only difference between a genotype and a phenotype is that in the latter all connections have weights assigned to them, the weights do not play a role in determining which genes are transferred to the offspring during crossover. Connections are inherited by sampling the

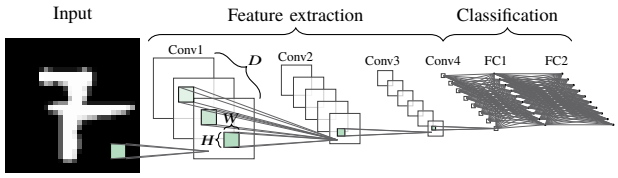


Fig. 1. A typical convolutional NN architecture. The first layer convolves the input with a set of kernels (filters) to obtain a shift-invariant response map of the input. The kernels in each convolutional layer have dimensionality $D \times W \times H$, where W and H are the width and height of the kernel and D is the number of input channels (i.e., the convolution 'depth'). By convention, all kernels in a convolutional layer have the same dimensions, so the kernel size effectively becomes an attribute of the layer itself. In addition, the number of channels is taken as the number of input maps in the previous layer. In the example above, D represents the number of channels for kernels in layer $Conv2$. The classification part consists of one or more fully connected layers.

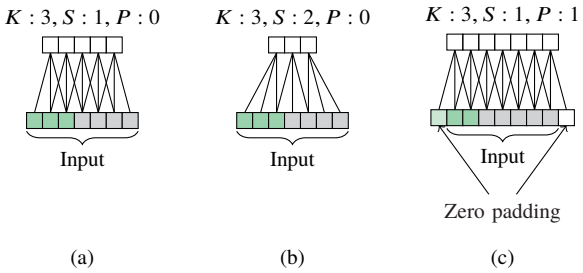


Fig. 2. An illustration of a convolution operation in one dimension. A kernel with size K is slid across the input, and the dot product of the kernel and the patch of the input that it covers is computed as the response of the kernel for that patch. The stride S determines how many tiles the kernel is shifted by at each step, and the padding P determines the offset of the convolution operation (i.e., whether or not it is aligned with the edge of the input). Both S and P determine the size of the response. (a~c) Kernel responses for different values of K, S and P.

[2]In Cortex, a group of networks belonging to the same species is referred to as a *population*, and a group of two or more coexisting populations is referred to as an *ecosystem*.

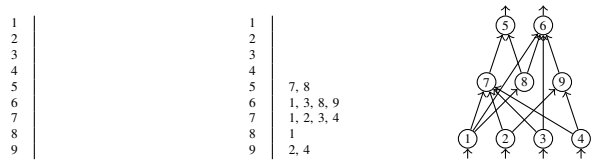


Fig. 3. (a) A Cortex genome (an ordered array of nodes), (b) a corresponding genotype (a genome with added inward connections, but no weights assigned to them) and (c) a complete phenotype (a NN with weights assigned to all connections).

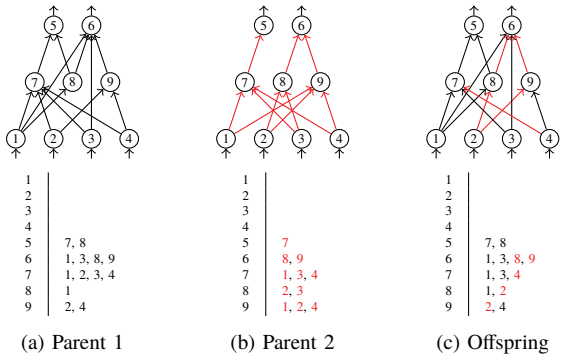


Fig. 4. Crossover operation between unstructured parent networks with a matching number of nodes. (a, b) Parents participating in the crossover and (c) the resulting offspring. Nodes in the genotype are ordered, which aids with topology matching during crossover.

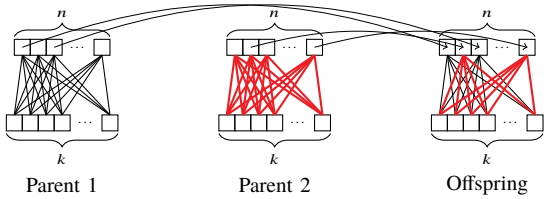


Fig. 5. Crossover between layered networks with matching number of nodes in each layer is guaranteed to preserve the dimensionality of each layer's input. This means that nodes can be transferred together with all of their input connections from the parents to the offspring without modification. In the case of convolutional layers, each kernel represents a node that can be inherited from the parent. Therefore, being able to manipulate individual kernels in a convolutional layer is essential (cf. III-E).

parent genotypes with probabilities proportional to the parents' relative fitness values (cf. (2)). If a connection exists in both parents, the offspring is more likely to inherit the connection weight from the fitter parent, whereas if a connection exists in only one parent, that parent's relative fitness is used as a probability to check if the connection should be inherited or skipped altogether.

In Cortex, the number of nodes in the genome is used as the sole criterion for speciation. A welcome side effect of this speciation technique is that the ecosystem can be initialised with more than one species from the onset, which facilitates exploration and promotes diversity.

Arguably, the most beneficial aspect of matching network topologies based on the ordered number of nodes is in regard to crossover, which is guaranteed to produce functional offspring containing *only* genes from the parents, without the need to introduce new connections or prune existing ones (Fig. 4). Although the benefits of this are less obvious for unstructured networks since in general they do not impose any restrictions on the dimensionality of the fan-in and the fan-out of individual nodes, it becomes important for layered networks, where every layer expects the input to have a certain shape and size. This issue was considered, for example, in [18], where the fan-in dimensionality of layer modules of incompatible size was adjusted in order to produce a functional model.

It should be noted that in Cortex weights are inherited unaltered from the parent networks rather than being initialised at every generation. In this context, the evolution mode in Cortex is epigenetic (or Lamarckian) since offspring effectively inherit traits that the parent networks have acquired during their lifetime.

III. Evolving deep convolutional models

In this study, the same topology matching and speciation scheme is applied to layered networks, which ensures that crossover between two layered networks preserves the input dimensionality of nodes and layers. This allows nodes together with *all of their input connections* to be treated as indelible genes which can be transferred unaltered from the parents to the offspring (Fig. 5). In layered networks, layers can be viewed as chromosomes containing a number of genes (nodes). Two species are considered identical if they have the same number of layers of each type (convolutional and FC), each containing the same number of nodes. However, when speciation is not used, it may become necessary to add or remove nodes (and corresponding input connections) to ensure that the input dimensionalities of layers in the offspring are correct, which may result in a lower overall fitness of new offspring. To test this intuition, the experiments presented in IV were performed with speciation enabled and disabled.

In the proposed framework, a population of deep CNNs is initialised and subsequently evolved over a number of generations. A single generation consists of several procedures, starting with training and evaluating all networks, followed by calibration, evolution (mutation and crossover), and finally culling. These procedures are described in more detail below.

A. Ecosystem initialisation

With speciation enabled, the initial networks in the ecosystem are distributed evenly among the initial number of species, where each network in a species is initialised according to the species genome. The first species always has a minimal genome (containing only an output layer looking directly at the input). Each subsequent species is generated from an isolated network (i.e., a randomly generated network which does not belong to any species) which is mutated randomly until its genome does not match that of any of the existing species. The isolated network's genome is then used to generate the next

species, and so on until the preset number of initial species is reached.

With speciation disabled, the initial ecosystem is essentially treated as a single species regardless of the genome. Networks are generated one at a time with a minimal genome (just an output layer), and a random mutation is applied to each network to promote initial diversity.

The shape of each kernel is initialised by sampling a weighted distribution of kernel shapes. Each shape is assigned a probability weighting $p_{w,h}$ inversely proportional to the area of the corresponding kernel computed as the product of its width and height dimensions d_w and d_h (not considering the number of channels):

$$p_{w,h} = \exp(-\prod_{w,h} d_w d_h) \tag{1}$$

For example, a kernel of size 3×3 would have a probability weighting of $\exp(-9)$ (≈ 0.00012). The same procedure is used to initialise the stride of convolutional layers. The motivation for this initialisation method is to ensure that most networks start with minimal kernels and strides and increase them over the course of the evolution. Kernel and stride mutations are described in more detail in III-E.

B. Training and evaluation

All networks are trained with standard BP with Adadelta as the optimiser by default, which is chosen because it does not require the learning rate to be set manually. The classification accuracy on the test set is used as the absolute fitness.

C. Calibration

Before the evolution step, the fitness values and complexity of all networks are scaled to fall between 0 and 1 by computing the mean μ_f and standard deviation σ_f of the fitness values of networks in the species (or the entire ecosystem if speciation is disabled). Then, the relative fitness $f_r(n_i)$ for network n_i is computed from its absolute fitness $f_a(n_i)$ as follows:

$$f_r(n_i) = g\left(\frac{f_a(n_i) - \mu_f}{\sigma_f}\right) \tag{2}$$

where g denotes the logistic function. Scaling the fitness to a value between 0 and 1 enables the relative fitness to be used as a probability for various random operations.

D. Mutation

Direct addition and removal of layers (both convolutional and fully connected) and nodes (in convolutional layers, a node is defined as a single kernel) is allowed, as well as changing the size of individual kernels and the stride of convolutional layers. For the kernel size and stride parameters, mutation is applied to a randomly selected dimension with probability as outlined below.

Structural mutations (adding or removing a node or a layer, resizing a kernel or resizing the stride parameter of a convolutional layer) are performed by sampling mutation types from a weighted probability distribution. The probability weighting of each mutation type is inversely proportional to an estimate of how many connections in the network the mutation would affect. For stride resizing, we use an estimate of how many *output* nodes in convolutional layers would be affected:

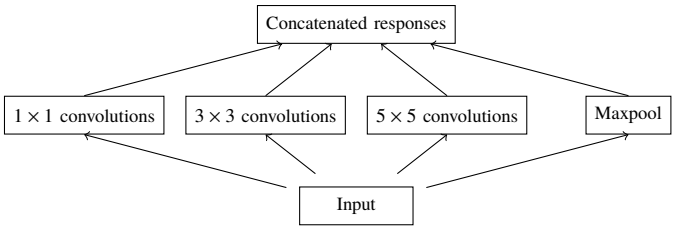


Fig. 6. A high-level representation of an Inception module. The input is convolved with kernels of multiple sizes, and the results are concatenated before being fed into the next layer.

$$p_{layer} \propto \frac{1}{\mu_c(\mu_n + \sigma_n)} \tag{3}$$

$$p_{node} \propto \frac{1}{\mu_c} \tag{4}$$

$$p_{stride} \propto \frac{1}{N_{cl}\mu_o} \tag{5}$$

$$p_{kernel} \propto \frac{1}{N_k \overline{A_k}} \tag{6}$$

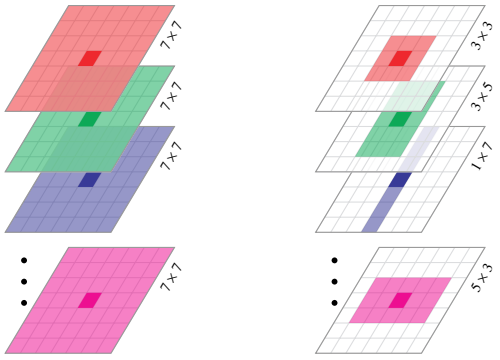
where μ_c is the mean number of connections per node, μ_n and σ_n are the mean and standard deviation of the node count per layer, N_{cl} is the number of convolutional layers, μ_o is the mean number of output nodes per convolutional layer, N_k is the mean number of kernels[3], and $\overline{A_k}$ is the mean kernel area ($W \times H$) in the network's convolutional layers. All of these probability weightings are computed by using statistics only for the network being mutated.

E. Evolving convolutional layer parameters

As mentioned above, it is usually assumed that the kernel size is a property of the layer rather than individual kernels. In other words, all kernels in the same layer usually have the same shape and size. However, the recently proposed Inception module architecture (Fig. 6) [32] breaks this trend by using kernels of different size (1×1, 3×3 and 5×5 in the original paper) to convolve the input of some of the layers, and the outputs of all the kernels are concatenated to produce the final output of the layer. The GoogLeNet implementation of the Inception architecture won the 2014 ILSVRC[4] challenge with a top-5 error rate of 5%.

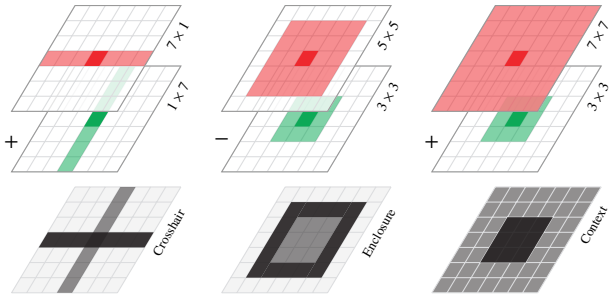
[3]The experiments below were conducted with the mean number of all nodes (N_n) instead of N_k due to an oversight. However, in practical terms this did not significantly change the ratio of the corresponding weights in the weighted distribution.

[4]ImageNet Large Scale Visual Recognition Competition



(a) (b)

Fig. 7. (a) A layer containing square kernels with the same size (7×7 tiles). The kernel size is essentially a property of the layer. (b) A layer with kernels of various shapes and sizes stacked along a line passing through the central element (marked as a darker tile).



(a) (b) (c)

Fig. 8. (a) A pair of two long skinny kernels spanning orthogonal dimensions can detect intersecting (crosshair-shaped) features. (b) Detecting enclosures within a single layer is possible by subtracting the activations of two kernels of different shapes. (c) Large kernels can provide contextual information for features extracted by smaller kernels. This information becomes immediately available to the next layer.

Inspired by the concept of convolving the input with kernels of various sizes, we go one step further and consider convolutional layers containing kernels of different *shapes*, where the shape is taken as a property of individual kernels rather than the entire layer. Below, we use kernel *shape* instead of kernel *size* to indicate that the kernel might not be square. Specifically, under the mild assumption that kernels have an odd number of tiles in all dimensions (e.g., 3×7 for a 2D kernel), all kernels in a layer can be stacked along a line passing through their central element (Fig. 7).

The motivation behind this design is that having kernels with different shapes in the same layer allows the layer to readily 'notice' certain features that would otherwise be masked if all kernels had the same shape. For instance, long skinny kernels can act as sharp edge detectors without pollution from neighbouring pixels, and pairs of such kernels that extend in different dimensions can act as detectors for crosshair-shaped features (Fig. 8a). Furthermore, pairs of kernels that differ by one or more tiles in each dimension (for example, 7×7 and 5×5) would effectively act as enclosure detectors for features that surround the central receptive field shared by the two kernels (Fig. 8b). In the same line of thought, large kernels can provide the next layer with contextual information for the finer, sharper features detected by smaller kernels (Fig. 8c). In other words, combinations of small and large kernels can reduce the confusion arising from having a lot of fine-grained features obtained from small kernels in the previous layer which cannot be easily associated with each other for lack of longer-range dependency information from one layer to the next.

Viewing the kernel shape as a property of each kernel allows us to introduce a new kernel shape mutation operator which affects individual kernels rather than the entire layer. Although the implementation of such irregularly shaped layers is fairly straightforward (it requires keeping the kernels as separate tensors and stacking them into a single weight tensor with dimensions equal to the largest dimensions of those kernels along the line passing through the central tile right before evaluation), none of the existing deep learning libraries support such operations out of the box due to the assumption that the layer has a consistent shape. For example, in the case of $2D$ convolution, a layer is usually implemented a $4D$ tensor of shape $N \times D \times W \times H$, where N is the batch size, D is the number of channels, and $W \times H$ is the shape (width and height) of the convolutional kernel. Consequently, it was necessary to use a library which is flexible enough to allow for such a modified evaluation procedure to be implemented while being efficient enough to minimise the impact of the extra stacking operation (cf. III-H).

Kernels are mutated by either growing or shrinking either the width or height of a single kernel by 2. For example, a 3×5 kernel could grow into a 5×5 or a 3×7 kernel by adding a 1×5 or a 3×1 block of tiles on both sides of the existing kernel along the width or height dimension, respectively. In this way, the functionality of the central part of the kernel is preserved while ensuring that the width and height remain odd so that all kernels in the same layer can be stacked along a line passing through the central tile of each kernel. All kernels always span the full depth of the input (i.e., all input channels), and the number of channels does not participate in mutations. Kernels are not allowed to grow larger than half of the input in width or height, and the minimal kernel size is naturally 1×1.

The stride of convolutional layers is evolved by growing or shrinking the stride parameter by 1 in the width or height dimension. Stride mutations are very disruptive as they do change the output size of the mutated layer and all layers above it, and for that reason stride mutations are much rarer that kernel mutations. The padding parameter is not mutated. Instead, it is set to half of the size of the largest kernel in each dimension (rounded down) to ensure that the size of the response of the convolutional layer is the same as the size of the input regardless of the shapes and sizes of the kernels (cf. Fig. 2c). This is important because otherwise kernel mutation might have a far-reaching effect similar to stride mutation.

F. Crossover

The procedure outlined in II-B is used for crossover, regardless of whether speciation is enabled or disabled. With speciation disabled, crossover proceeds by iterating over the layers in the two parents and subsequently iterating over nodes in each layer. If one of the networks happens to contain fewer layers of a particular type than the other, the excess layers of that type from the larger parent are inherited unaltered with a probability proportional to the fitness of the larger parent. The same principle is applied to nodes in each layer in case of layer size mismatch.

With speciation enabled, crossover is restricted to networks within the same species. At the crossover step, all networks in a species are selected to be parents with probability proportional to their respective relative fitness values (cf. III-C). This is done in order to give all networks a chance to reproduce while ensuring that networks with higher fitness would have a better chance to do so. With speciation disabled, any network can participate in crossover with any other network in the ecosystem. In this case, we adopt the following measure for the similarity of the genomes of two networks n_i and n_j:

$$s_{n_i,n_j} = \frac{O_{i,j}}{N_i + N_j - O_{i,j}} \tag{7}$$

where $O_{i,j}$ is the total intersection and N_i and N_j are the total number of nodes in the genomes. The total intersection $O_{i,j}$ is computed as

$$O_{i,j} = \sum_k L_k(n_i) \cap L_k(n_j) \tag{8}$$

where $L_k(n_i) \cap L_k(n_j)$ is the intersection of the k^{th} layer of the same type (i.e., convolutional or fully connected) in networks n_i and n_j in terms of number of nodes. For networks with the same number of layers of each type and the same number of nodes in each layer, this similarity measure is 1, which coincides with the case where speciation is enabled. Since $s_{n_i,n_j} \in [0, 1]$, it can be used as a probability for crossover.

G. Culling

Once the ecosystem grows beyond the preset limit as a result of crossover, a culling procedure takes place to reduce the ecosystem size. New offspring are guaranteed to survive, as well as the champion for each species (or the ecosystem champion with speciation disabled). All other networks are sampled from a weighted distribution with probability weighting $p_{cull}(n_i)$ for network n_i computed as follows:

$$p_{cull}(n_i) = \frac{age(n_i)}{f_r(n_i)c_r(n_i)} \tag{9}$$

where $age(n_i)$ is the age of network n_i in terms of epochs. The culling continues by selecting networks one by one until the ecosystem size limit is reached.

H. PyCortex NE platform

We developed a NE platform (PyCortex) which implements the above framework, including a mutation operation capable of altering the size of individual kernels as outlined in III-E as well as other common mutation operations (adding and removing nodes and entire layers and mutating the stride of convolutional layers). In essence, PyCortex provides a convenient interface to an established deep learning platform (PyTorch[5]) and can be used for direct evolution of both regular and convolutional deep NNs by abstracting the computational details of the crossover and mutation operations. Each evolved network is a valid model which can be evaluated directly in PyTorch, harnessing all the effort that has been invested into making the platform efficient and flexible. At present, PyCortex employs a hybrid strategy where the network structure is evolved while weights are optimised with BP. However, PyTorch provides easy access to all learnable parameters in a model, which paves the way to testing alternative weight optimisation algorithms, such as evolutionary strategies. The proposed platform is released as an open-source project[6] to facilitate research in NE. We hope that as it matures it can serve as a standard tool for prototyping and benchmarking novel NE algorithms.

IV. Experiments

We conducted experiments on several image classification tasks to test the feasibility of the proposed framework with respect to deep convolutional models. Following the insight in [34], the layer types were limited to convolutional and fully connected, without pooling layers. All experiments were performed with and without speciation to test whether crossover in the case without speciation would result in lower offspring fitness (cf. Fig. 5). All other configuration options were identical across the experiments. With speciation enabled, ecosystems were initialised with 8 species, and a hard limit of 16 species was used in all experiments.

The experiments were conducted with initial and maximal ecosystem size of 64 and 111, respectively[7]. Connection weights were drawn from a normal distribution with mean 0 and standard deviation of 0.1, both during ecosystem initialisation and when any new connections were added through mutations.

Two important points about the training procedure are worth emphasising.

- At each generation (training epoch), each network in the ecosystem was trained on a **random 10% subset** of

[5]PyTorch was chosen for its flexible interface which allows network evaluation graphs to be generated at runtime, which was essential for the efficient implementation of convolutional layers with varying kernel sizes.

[6]PyCortex repository on GitLab.

[7]The number 111 was an artefact of the cluster configuration. The goal was to ensure that each network had a dedicated CPU core for evaluation. Each node on the particular cluster we used contains 28 cores, and four nodes were used for each experiment, raising the total core count to 112. However, one core was reserved for the master MPI process, bringing the total number of available cores to 111.

the training data, after which it was evaluated on the entire test set. The accuracy on the test set was used as the absolute fitness of that network. The decision to use a small random portion of the data for training was motivated by the aim to examine whether crossover can transfer learned features to the offspring in an epigenetic evolution scenario. For that purpose, all new offspring were evaluated on the test set prior to commencing training with BP.

- We used a relatively large training batch size of 128. Large training batches tend to produce solutions which converge to 'sharp' minima [35], with a negative impact on generalisation. Hence, it was of particular interest to examine whether the perturbations introduced by the mutation and crossover operations would still allow the networks to generalise well even with a large batch size.

A. Datasets

Four commonly used datasets (MNIST [36], SVHN [37], Fashion-MNIST [38]) and CIFAR-10 [39]) were used in the experiments. All datasets have 10 classes. MNIST and SVHN contain images of digits (preprocessed and grayscale in the case of MNIST, natural RGB in the case of SVHN), while Fashion-MNIST and CIFAR-10 contain images of 10 different types of objects (preprocessed and grayscale in the case of Fashion-MNIST, natural RGB in the case of CIFAR-10). This allows us to compare potential differences arising from the use of colour information. Each experiment was run 10 times for 100 generations per run.

B. Results

One of the key advantages of using an established platform for evaluation is that all the tools available for that platform can be readily used to track and analyse the learning progress and other parameters. Specifically, we used the tensorboardX module for saving epoch data in TensorBoard log format directly from PyTorch, which has the added advantage of being able to plot the data in real time. The results for the highest fitness, average fitness and average offspring fitness before BP for all experiments are summarised in Table I (results for MNIST with speciation enabled are presented in Fig. 9).

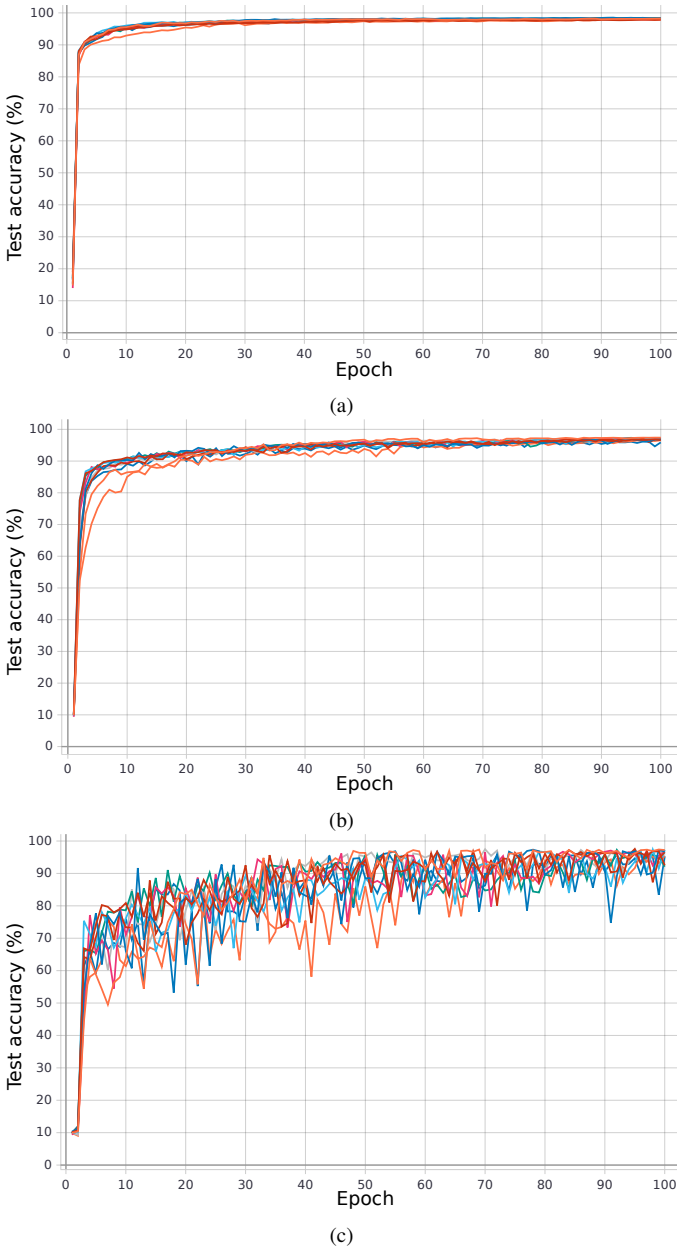


Fig. 9. Experiment results for the MNIST benchmark with speciation enabled. (a) Highest fitness, (b) average fitness and (c) average fitness of new offspring before BP.

V. Discussion

The evolved CNNs performed admirably on the MNIST dataset, reaching an overall high scores of 98.32% and 98.14% with and without speciation, respectively, whereas the highest score for the SVHN dataset was considerably lower (although still above 80%). This reflects the higher difficulty of classifying natural images versus pre-processed ones, as the objective is the same in both cases (classifying digits). The same trend can be observed in the case of Fashion-MNIST and CIFAR-10, where it is even more pronounced.

The results in Table I reveal surprisingly small differences for the highest recorded fitness between the experiments with and without speciation. In previous research, we employed speciation to protect mutated networks from being eliminated from the ecosystem before they had had a chance to optimise any new weights introduced by mutations. However, in that case weights were optimised by evolution, whereas in the above experiments the weights were optimised by BP, which likely reduces the importance of speciation for this particular purpose. Furthermore, the experiment logs (not shown) revealed that in the experiments with speciation disabled the parameter count increased steadily over the 100 epochs, whereas in those with speciation enabled it was essentially stagnant and even *decreased* in a number of runs. In all cases with speciation enabled, the logs also revealed a large number of failed structural mutations (addition and

Table I
HIGHEST VALUE, MEAN AND STANDARD DEVIATION RECORDED OVER 10 RUNS AFTER EPOCH 100 FOR THE HIGHEST FITNESS, AVERAGE FITNESS AND AVERAGE OFFSPRING FITNESS BEFORE BP FOR EACH EXPERIMENT.

Experiment	Highest fitness	Average fitness	Average offspring fitness before BP
MNIST (with speciation)	98.32% (98.13 ± 0.15)	97.14% (96.8 ± 0.4)	97.28% (95.6 ± 1.8)
MNIST (no speciation)	98.14% (97.98 ± 0.13)	95.31% (93.7 ± 1.1)	82.98% (72.5 ±10.7)
SVHN (with speciation)	82.84% (79.86 ± 2.06)	76.9% (70.2 ± 7.5)	77.25% (66.5 ± 8.0)
SVHN (no speciation)	81.20% (80.22 ± 0.89)	70.62% (66.5 ± 3.1)	55.53% (44.4 ± 7.4)
Fashion-MNIST (with speciation)	89.49% (88.54 ± 0.70)	87.89% (85.3 ± 1.4)	88.18% (82.0 ± 4.3)
Fashion-MNIST (no speciation)	88.88% (88.44 ± 0.36)	85.53% (83.3 ± 1.1)	83.61% (64.9 ± 7.8)
CIFAR-10 (with speciation)	55.52% (51.35 ± 3.44)	46.45% (43.9 ± 4.8)	45.94% (42.7 ± 5.1)
CIFAR-10 (no speciation)	54.4% (52.44 ± 0.85)	43.87% (41.9 ± 1.1)	35.8% (32.4 ± 1.8)

removal of nodes and layers) due to reaching the limit on the species count, which is likely the cause for the stagnation. In this regard, the champions in all experiments were relatively small (~10^5 parameters), which can at least partially explain the low scores obtained on CIFAR-10 and, to a smaller extent, SVHN and Fashion-MNIST. Nevertheless, for all four datasets, the average overall fitness and the average offspring fitness before BP at epoch 100 are higher in the cases with speciation than in those without, but more experiments are necessary in order to run a proper statistical analysis on the significance level of this difference. In future work, we plan to develop a more robust method for determining the mutation rate for complexifying mutations in order to evolve much larger models with potentially millions of parameters. We are also working on a dynamic speciation limit which is designed to altogether eliminate the need to set a hard species limit, allowing the number of parameters to increase more rapidly when speciation is enabled.

Perhaps the most satisfying part of the results is the average offspring fitness before BP, which represents the performance of new offspring evaluated *before any training*. As outlined in IV, all networks were trained on a random 10% subset of the training data at each epoch, which means that no single network ever saw the entire dataset. However, over the course of 100 epochs, practically all of the dataset would have been *collectively* seen by the ecosystem. The only way that this could be useful to new offspring would be if crossover could successfully preserve and transfer learned features by combining useful genes (nodes with all of their input connections) from the parents into the offspring. The results for the average offspring fitness before BP, which increased steadily over the 100 epochs in all cases (cf. Fig. 9c; a similar trend was observed in all other experiments), provides direct confirmation that this is in fact the case. There is also a clear difference in the results for the offspring fitness before BP between experiments with and without speciation, which confirms the intuition presented in II-B that speciation affects the fitness of new offspring by determining the way genes and chromosomes are matched during crossover.

VI. CONCLUSION

This study extended previous research on NE and demonstrated its applicability to the direct evolution of deep convolutional models. A new convolutional layer layout which allows kernels of different size and shape to coexist within the same convolutional layer was also proposed, and a corresponding mutation operator which can resize individual kernels was introduced. A detailed analysis and visualisation of the kernels in evolved CNNs will be presented in future work. Furthermore, the crossover procedure previously proposed for unstructured networks was extended to deep layered networks, and its feasibility was demonstrated through image classification experiments with evolved CNNs. Finally, a NE platform which implements the proposed framework, including crossover and kernel mutation operators for deep CNNs, was developed on top of an established deep learning library (PyTorch). This platform is released as an open-source project with the aim to provide a common framework for prototyping, evaluating and comparing NE algorithms.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions, which helped improve the readability and the overall quality of the paper. This research was undertaken with the assistance of resources and services provided by the National Computational Infrastructure, which is supported by the Australian Government.

REFERENCES

[1] A. P. Wieland, "Evolving neural network controllers for unstable systems," *International Joint Conference on Neural Networks*, IEEE, 1991, pp. 667–673.

[2] N. Richards, D. Moriarty, and R. Miikkulainen, "Evolving Neural Networks to Play Go," *Applied intelligence* (1998), pp. 85–96.

[3] X. Yao, "Evolving Artificial Neural Networks," *Proceedings of the IEEE* 9 (1999), pp. 1423–1447.

[4] F. J. Gomez and R. Miikkulainen, "Solving non-Markovian control tasks with neuroevolution," *International Joint Conference on Artificial Intelligence*, 1999, pp. 1356–1361.

[5] N. Hansen and A. Ostermeier, "Completely derandomized self-adaptation in evolution strategies," *Evolutionary computation* 2 (2001), pp. 159–195.

[6] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," *Evolutionary computation* 2 (2002), pp. 99–127.

[7] F. Gomez, J. Schmidhuber, and R. Miikkulainen, "Efficient non-linear control through neuroevolution," *ECML*, Springer, 2006, pp. 654–662.

[8] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: from architectures to learning," *Evolutionary intelligence* 1 (2008), pp. 47–62.

[9] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, "A Hypercube-Based Indirect Encoding for Evolving Large-Scale Neural Networks," *Artificial life* 2 (2010), pp. 185–212.

[10] J. Secretan, N. Beato, D. B. D'Ambrosio, et al., "Picbreeder: A Case Study in Collaborative Evolutionary Exploration of Design Space," *Evolutionary computation* 3 (2011), pp. 373–403.

[11] M. M. Khan, A. M. Ahmad, G. M. Khan, and J. F. Miller, "Fast learning neural networks using Cartesian genetic programming," *Neurocomputing* (2013), pp. 274–289.

[12] A. J. Turner and J. F. Miller, "NeuroEvolution: Evolving Heterogeneous Artificial Neural Networks," *Evolutionary Intelligence* 3 (2014), pp. 135–154.

[13] C. Igel, "Neuroevolution for reinforcement learning using evolution strategies," *Congress on Evolutionary Computation*, IEEE, 2003, pp. 2588–2595.

[14] J. Lehman and K. O. Stanley, "Improving evolvability through novelty search and self-adaptation," *Congress on Evolutionary Computation*, 2011, pp. 2693–2700.

[15] P. Verbancsics and J. Harguess, *Generative NeuroEvolution for Deep Learning*, 2013, arXiv: 1312.5355v1.

[16] E. Real, S. Moore, A. Selle, et al., "Large-Scale Evolution of Image Classifiers," *ICML*, 2017, pp. 2902–2911.

[17] R. Miikkulainen, J. Z. Liang, E. Meyerson, et al., "Evolving Deep Neural Networks" (2017), arXiv: 1703.00548.

[18] S. R. Young, D. C. Rose, T. Johnston, et al., "Evolving Deep Networks Using HPC," *Proceedings of the Machine Learning on HPC Environments*, 2017, 7:1–7:7.

[19] C. Fernando, D. Banarse, M. Reynolds, et al., "Convolution by Evolution: Differentiable Pattern Producing Networks," *GECCO*, 2016, pp. 109–116.

[20] L. Xie and A. L. Yuille, "Genetic CNN" (2017), arXiv: 1703.01513.

[21] B. Wang, Y. Sun, B. Xue, and M. Zhang, "Evolving Deep Convolutional Neural Networks by Variable-Length Particle Swarm Optimization for Image Classification," *Congress on Evolutionary Computation*, 2018, pp. 1–8.

[22] M. Suganuma, S. Shirakawa, and T. Nagao, "A Genetic Programming Approach to Designing Convolutional Neural Network Architectures," *IJCAI*, 2018, pp. 5369–5373.

[23] A. Nayebi, D. Bear, J. Kubilius, et al., "Task-Driven Convolutional Recurrent Models of the Visual System," *NeurIPS*, 2018, pp. 5295–5306.

[24] T. Desell, "Developing a Volunteer Computing Project to Evolve Convolutional Neural Networks and Their Hyperparameters," *2017 IEEE 13th International Conference on e-Science*, 2017, pp. 19–28.

[25] Y. Sun, B. Xue, and M. Zhang, "Evolving Deep Convolutional Neural Networks for Image Classification" (2017), arXiv: 1710.10741.

[26] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning Both Weights and Connections for Efficient Neural Networks," *International Conference on Neural Information Processing Systems*, 2015, pp. 1135–1143.

[27] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing Neural Network Architectures using Reinforcement Learning," *International Conference on Learning Representations*, 2017.

[28] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning" (2017), arXiv: 1703.03864.

[29] A. Hadjiivanov and A. Blair, "Complexity-based speciation and genotype representation for neuroevolution," *Congress on Evolutionary Computation*, 2016, pp. 3092–3101.

[30] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE* 11 (1998), pp. 2278–2324.

[31] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Conference on Neural Information Processing Systems*, 2012, pp. 1106–1114.

[32] C. Szegedy, W. Liu, Y. Jia, et al., "Going deeper with convolutions," *CVPR*, 2015, pp. 1–9.

[33] K. O. Stanley, "Efficient evolution of neural networks through complexification," PhD thesis, Department of Computer Sciences, University of Texas, 2004.

[34] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. A. Riedmiller, "Striving for Simplicity: The All Convolutional Net" (2014), arXiv: 1412.6806.

[35] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima" (2016), arXiv: 1609.04836.

[36] Y. LeCun and C. Cortes, "MNIST handwritten digit database" (2010).

[37] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," *NIPS workshop on deep learning and unsupervised feature learning*, 2, 2011, p. 5.

[38] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms" (2017), arXiv: 1708.07747.

[39] A. Krizhevsky and G. Hinton, *Learning multiple layers of features from tiny images*, tech. rep., University of Toronto, 2009.