A Graph-Based Encoding for Evolutionary Convolutional Neural Network Architecture Design

Conference Paper · June 2019

DOI: 10.1109/CEC.2019.8790093

Yanan Sun
Sichuan University
29 PUBLICATIONS **331** CITATIONS
SEE PROFILE

Bing Xue
Victoria University of Wellington
229 PUBLICATIONS **3,421** CITATIONS
SEE PROFILE

Mengjie Zhang
Victoria University of Wellington
668 PUBLICATIONS **8,974** CITATIONS
SEE PROFILE

A Graph-Based Encoding for Evolutionary Convolutional Neural Network Architecture Design

William Irwin-Harris, Yanan Sun, Bing Xue, and Mengjie Zhang
School of Engineering and Computer Science
Victoria University of Wellington, P.O. Box 600
Wellington 6140, New Zealand
Emails: {william.irwin-harris,yanan.sun,bing.xue,mengjie.zhang}@ecs.vuw.ac.nz

Abstract—**Convolutional neural networks (CNNs) have demonstrated highly effective performance in image classification across a range of data sets. The best performance can only be obtained with CNNs when the appropriate architecture is chosen, which depends on both the volume and nature of the training data available. Many of the state-of-the-art architectures in the literature have been hand-crafted by human researchers, but this requires expertise in CNNs, domain knowledge, or trial-and-error experimentation, often using expensive resources. Recent work based on evolutionary deep learning has offered an alternative, in which evolutionary computation (EC) is applied to automatic architecture search. A key component in evolutionary deep learning is the chosen encoding strategy; however, previous approaches to CNN encoding in EC typically have restrictions in the architectures that can be represented. Here, we propose an encoding strategy based on a directed acyclic graph representation, and introduce an algorithm for random generation of CNN architectures using this encoding. In contrast to previous work, our proposed encoding method is more general, enabling representation of CNNs of arbitrary connectional structure and unbounded depth. We demonstrate its effectiveness using a random search, where 200 random CNN architectures are generated and then each is trained using only 10% of the CIFAR-10 training data (i.e., using 4,500 training images and 500 validation images). The three best-performing CNNs are re-trained on the full 50,000 training images. The results show that the proposed representation and initialisation method can achieve promising accuracy compared to manually designed architectures, despite the simplicity of the random search approach and the reduced data set. We intend that future work can improve on these results by applying evolutionary search using this encoding.**

I. Introduction

Deep learning [1] has achieved great success in image classification [2], with convolutional neural networks (CNNs) becoming the dominant choice of model. Advances in deep learning for image recognition have been driven in large part by the introduction of new CNN architectures: following the introduction of CNNs with LeNet5 [3], other successful architectures have included AlexNet [2], VGG [4], GoogleNet [5], ResNet [6] and DenseNet [7].

Yet designing CNNs is still a challenging problem: it requires not only expertise from machine learning researchers familiar with CNNs, but also knowledge of the specific problem domain in order to achieve the best results [8]. Typically, substantial trial-and-error experimentation is also required. Hence designing new CNNs for a specific application is often time-intensive and costly, due to the need for human expertise. Moreover, even if a decision is made to use an already-developed (e.g. published) CNN architecture for a real-world problem, this may not produce the best results, as the architecture has not been designed specifically for the target application and its corresponding data. In particular, for a small amount of training data, a relatively shallow CNN may give the best results. Conversely, if there is a large amount of training data available, the best results may be obtained by a very deep CNN.

Because of these challenges associated with manually selecting a CNN architecture, automatic techniques for deriving new architectures have been a topic of recent work. Existing work in this area has generally employed either of two key architecture search strategies. One of these is reinforcement learning, which is the approach taken by methods such as neural architecture search [9], MetaQNN [10] and Block-QNN [11]. These methods have achieved good results, but have generally required extensive computational resources. The other key strategy employed is based on evolutionary deep learning, in which evolutionary computation methods are employed to derive new deep structures. Work in this category includes the Genetic CNN method [12], the large-scale evolution method [13], the hierarchical representations method [14], the Cartesian genetic programming–based method [15], the CNN-GA method [16] and the GP-CNAS method [17].

Evolutionary architecture search algorithms have achieved good results on benchmark data sets, including CIFAR-10 and CIFAR-100 [18], demonstrating the potential of this approach. However, several limitations remain; we suggest these are largely due to the choice of encoding strategy, which is a key component in evolutionary deep learning. Both the Genetic CNN [12] and CGP-CNN methods [15] limit the final depth of the evolved CNN, such that a maximum depth must be specified by the user before beginning the evolutionary process. For CGP-CNN [15], this is due to the use of the standard Cartesian genetic programming encoding of a grid with a fixed number of rows and columns ($N_r \times N_c$). Another limitation of CGP-CNN [15] is that a crossover operator is not used; the algorithm evolves architectures using point mutation only. In the case of Genetic CNN [12], the maximum CNN depth is also fixed before the algorithm is run, due to the encoding strategy of a fixed-length binary string.

More broadly, these limitations can be described as showing the algorithm to be *semi-automatic* rather than fully automatic; that is, in the previous two examples, user input is still required to specify the maximum depth for the encoding. A similar requirement also exists for the hierarchical representation method [14], in which the user must predetermine the number of levels in the hierarchical representation, and also the number of nodes in the graph at each level.

Although the CNN-GA [16] approach is fully automatic, as it uses an encoding of a linked list of building blocks that may be grown to arbitrary depth during the evolution, the connectional structure of the architectures it generates is restricted. Each building block in the linked list encoding is either a pooling layer or a 'skip layer', where a skip layer includes two convolutional layers and a skip connection [16]; this design is modelled on ResNet [6]. This search space achieves good results, but means that CNN-GA [16] cannot generate CNN architectures of arbitrary graph structure. Notable architectures that use a complex graph structure that differs from ResNet [6] include DenseNet [7] and the Block-QNN-S cell designed by the Block-QNN reinforcement learning approach [11].

Our goal in this paper is to address these limitations by developing a new encoding strategy that can represent CNN architectures of arbitrary graph structure and with unbounded depth. The key contributions of this paper are as follows:

1) We propose a novel encoding strategy, intended to be applied in evolutionary deep learning, in which each CNN architecture is represented as a directed acyclic graph. Compared to other representations designed for evolutionary search, we apply significantly fewer constraints to the search space. We believe this may enable search algorithms to derive novel structures, such as new applications of skip connections.
2) We introduce a method for randomly generating architectures according to the proposed encoding, such that the method can be used either as the population initialisation component of an evolutionary algorithm, or on its own in a random search. Our proposed initialisation method is designed to apply minimal restrictions to the search space, while avoiding generation of architectures that are invalid or evidently suboptimal.
3) The effectiveness of this encoding strategy and the associated initialisation method is demonstrated using a simple random search, using only 10% of the final training data during the search process. We show that even with a large search space, limited training data during search, and a simplistic search algorithm, the representation is able to achieve promising accuracy.

II. Background

In this section, we provide a brief review of the core components of CNNs, and in particular the use of skip connections.

A. Standard CNN Layers

The two key components of CNNs are convolutional layers and pooling layers. As this paper focuses on image recognition, we consider the case of 2D convolution kernels. For the 2D situation, the output of each layer can be described as a $C \times M \times N$ array, where C is the number of channels, and M and N are the width and height of a 2D array. Each such 2D array is often referred to as a feature map. The input of one layer can be taken directly from the output of a previous layer, or constructed by an operation such as summation or concatenation; these two operations are discussed in the next section. In addition, a layer may be connected directly to the image pixels, in which case the number of input channels is generally three (for the red, green and blue components).

A convolutional layer produces a stack of output feature maps by applying convolution kernels, which are most often square matrices of odd dimensions; typical sizes are 3×3, 5×5 and 7×7. A convolution kernel is applied by performing element-wise multiplication of the kernel values with the corresponding array elements from the input (either the input feature map or image), and then summing these products to produce the output of the kernel. During network training, each convolutional layer learns the values of its convolution kernels, where the number of kernels is determined by the dimensionality of the input data and the number of output feature maps. The number of output feature maps is a parameter of the layer.

Pooling layers are applied similarly to convolutional layers, but rather than using convolution kernels, an operator is applied to aggregate input data over a spatial area; typically this is either the arithmetic mean (average pooling) or maximum (max pooling). Most often, pooling layers are used for spatial dimensionality reduction; the number of channels is unchanged.

CNNs may optionally include one or more fully-connected (dense) layers at the end of the CNN structure, immediately prior to the output layer. These layers follow the standard structure of a multilayer feed-forward neural network, where each node in the layer is connected to every node in the previous layer. However, in our proposed encoding method, we avoid the use of fully-connected layers. This is consistent with the approach taken in related works [15], [16]; fully-connected layers are prone to overfitting due to their dense connections [19].

Hence the task of encoding a CNN architecture can be described as that of specifying the number of layers, their connectional structure, and the type and parameters of each layer.

B. Skip Connections

A key technique in recent CNN architectures has been *skip connections* (also known as shortcut connections). The most straightforward CNN architecture consists of a linear sequence of layers, where each layer takes its input directly from the previous layer. However, with the inclusion of skip connections, a layer's input may be specified as the summation of the immediate previous layer's output and also the output of a layer earlier in the sequence. The effectiveness of skip connections has been highlighted by their successful application in ResNet [6]. In ResNet, the structure is based on a repeating sequence of 'residual blocks' (each of which contains two convolutional layers and a skip connection) [6]. The benefits of skip connections in alleviating the vanishing gradient issue (which is common in deep neural networks) has been discussed in [20], and their effectiveness in deep learning has also been experimentally shown in [21].

The concept of skip connections can be generalised by considering CNNs as directed acyclic graphs of potentially arbitrary structure. In this case, we consider that the input to a convolutional or pooling layer may be the output of any other layer, or the summation or concatenation of the outputs of other layers, with the requirement that no cycles are formed. The summation operator performs the element-wise addition of two $C \times M \times N$ arrays; i.e., the two multidimensional arrays must have identical spatial dimensions and numbers of channels. The concatenation operator joins the channels of a $C_1 \times M \times N$ and a $C_2 \times M \times N$ array to produce an output array of size $(C_1 + C_2) \times M \times N$. As with summation, the spatial dimensions of the two arrays must be equal for concatenation.

III. The Proposed Encoding Strategy

This section discusses the proposed CNN encoding method, and presents an algorithm for random initialisation of CNN architectures described by the encoding.

A. Encoding Strategy Overview

As discussed earlier, the components of a CNN can be considered as a directed acyclic graph. In this subsection, we detail our proposed graph-based encoding. A general graph can be described as an ordered pair $G = (V, E)$ with vertices V and edges E. For a directed acyclic graph, each edge is associated with a direction, and there is a restriction that there is no path from a node v to itself. For our CNN architecture encoding, we note the further restriction that there is always *precisely one* node r such that there are no edges ending at r (but there may be edges beginning at r). We term this the root node, which represents the output of the CNN.

In this encoding, each node in the graph corresponds to either a building block, an operator, or a reference to the input data. We assign each node a type, which is one of SUM, CONCAT, CONV, MAX, AVERAGE and INPUT. In addition, CONV nodes also have a $feature\_maps$ attribute, which specifies the integer number of channels in the output of the convolutional layer. In this encoding, we choose a convention that an edge from node n_1 to node n_2 represents n_1 taking its input data from n_2. This is *opposite* to the direction of data flow in the network. That is, edges are directed from the network's output layer towards the input data (shown in the left-hand side of Fig. 1). This convention is arbitrary, but is convenient for the description of algorithms acting on this structure.

Each node type has a fixed *arity*, which specifies the required number of edges beginning at that node. Note that a node may have any number of edges ending at that node. Nodes of type INPUT are leaf nodes and so have arity 0; we define SUM and CONCAT nodes to have arity 2; and all other

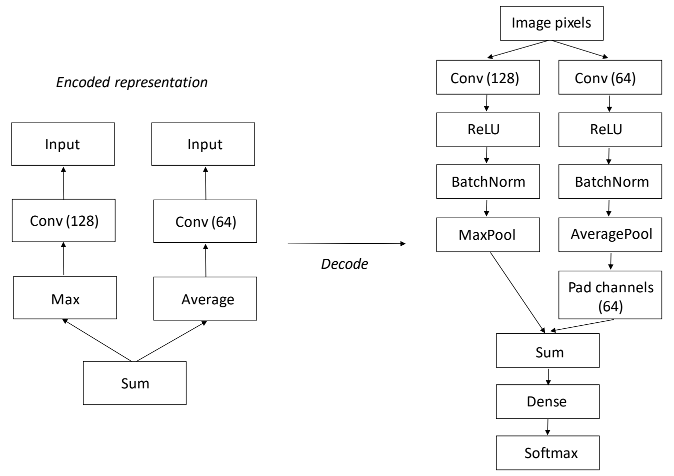

Fig. 1. An example of an encoded representation and its corresponding decoded CNN architecture.

nodes have arity 1. Summation and concatenation of arbitrary numbers of arrays can still be represented by chaining SUM and CONCAT nodes.

B. Representation and Decoding Details

In this subsection, we provide further details about the graph structure, and describe the decoding method used to transform this representation (genotype) into the final CNN architecture (phenotype). Fig. 1 gives an example of the encoded representation, and shows its corresponding decoded CNN architecture.

First, we detail the available node types, and justify these design choices. A CONV node represents a 'convolutional building block' in the final CNN, which we define as consisting of a convolutional layer with a stride of 1, a ReLU activation function [22], and a batch normalisation [23], in that order. This design is inspired by the ConvBlock used in the CGP-CNN method [15], and the use of batch normalisation and ReLU components are also consistent with recent hand-crafted CNNs [6], [7]. We use the 'same' padding mode and the 1×1 stride, so that the image dimensions do not change after applying convolution. The number of feature maps is configurable and is a parameter of the CONV node. In contrast, we use a fixed filter (kernel) size, which is always 3×3. This choice is made because 3×3 is a very commonly used filter size in manually designed CNNs such as ResNet [6] and DenseNet [7]; in addition, it has been suggested that filters of other sizes can be replaced by stacks of 3×3 filters without losing representational ability [24]. Thus, using a fixed filter size acts to reduce the search space.

The MAX and AVERAGE node types represent max and average pooling layers, respectively. We use a kernel size of 2 and stride of 2 for both max and average pooling layers, so that these layers always halve the spatial dimensions of the input data. Greater reductions in dimension can then be achieved by stacking these pooling layers. The INPUT node type is used to reference the current image (or minibatch of images) from the training or test data to which the CNN is being applied. Conceptually, only one INPUT node is required; however, it can be convenient for the algorithm implementation to allow multiple INPUT nodes, all of which are considered equivalent.

The SUM and CONCAT nodes represent the summation and concatenation operations, respectively. During the decoding step, if the dimensions of the two inputs to either SUM or CONCAT nodes are not equal, a max pooling layer is added to downsample the larger feature map, such that the dimensions of the two feature maps become equal. This is based on the procedure used in [15] and [16].

When decoding a SUM node, if the number of channels of the two inputs differ, we add padding channels containing all zeros to the input with the smaller number of channels, so that the number of channels is equal (see Fig. 1). This approach is based on a method discussed in the ResNet paper [6], which also suggests 1×1 convolutions as an alternative for adjusting channel dimensionality. For this decoding method, we choose to use zero-padded channels as this technique results in fewer parameters compared to 1×1 convolutions.

To construct the final CNN architecture from the encoded representation, we transform each node in the graph into its corresponding CNN layers, as detailed above. Finally, a fully-connected (dense) layer with a number of output nodes equal to the number of image classes is added to the end of the CNN. A softmax activation function is then added as the final component of the CNN.

C. Initialisation Algorithm Overview

In this subsection, we present an algorithm for random generation of a CNN architecture represented by the proposed encoding strategy. The overall framework of this method is given in Algorithm 1.

First, we randomly choose an integer depth d (line 1), which we refer to as the *target depth* within this initialisation algorithm. This allows for CNNs of a variety of depths when generating a population, which may increase the likelihood of obtaining an architecture with an appropriate depth for the target data set. The algorithm then uses a standard first-in, first-out (FIFO) queue to construct the graph in a breadth-first order. A random node is generated as the root (line 6), and then the graph is iteratively 'grown' in depth. For each node in the queue, the algorithm creates a number of 'child' nodes equal to the arity of the node (lines 15-32). Each item in the queue contains both a node and also the level ℓ of the node; the level is equal to the minimum number of edges from the node to the root. The level information is used to avoid generating nodes that exceed the target depth determined earlier. Nodes with a level $\ell = d - 1$ are constrained to have only INPUT nodes as children; nodes of type INPUT are leaf nodes, which prevents the graph from growing further (lines 16-17). Otherwise, for $\ell < d - 1$, the algorithm will randomly generate new nodes for the parent node (lines 19-29). The set R (line 19) is used to determine node types that are disallowed, in order to avoid generating invalid or suboptimal architectures (detailed in Algorithm 2).

Algorithm 1: Framework of the Initialisation Method

```
1: $d \leftarrow$ Generate a random integer depth greater than zero
2: $Q \leftarrow$ A new FIFO queue
3: $S \leftarrow$ A new, empty multiset
4: $level \leftarrow 0$
5: $nonleaf \leftarrow$ False
6: Generate a random non-null node $r$ as the root, and add
   the pair $(r, 0)$ to $Q$
7: while $Q$ is not empty do
8:     $(n, \ell) \leftarrow$ Dequeue a node and level from $Q$
9:     $a \leftarrow$ Get the arity of $n$
10:    if $\ell \neq level$ then
11:        $nonleaf \leftarrow$ False
12:        $level \leftarrow \ell$
13:    end if
14:    $i \leftarrow 0$
15:    while $i < a$ do
16:        if $\ell + 1 = d$ then
17:            Add an edge from $n$ to a new INPUT node
18:        else
19:            $R \leftarrow$ Compute a set of disallowed node types
20:            $c \leftarrow$ Generate a random node restricted by $R$
21:            if $c = null$ then
22:                $S \leftarrow S \cup \{n\}$
23:            else
24:                Add an edge from $n$ to $c$
25:                if $type(c) \neq$ INPUT then
26:                    Enqueue the pair $(c, \ell + 1)$ to $Q$
27:                    $nonleaf \leftarrow$ True
28:                end if
29:            end if
30:        end if
31:        $i \leftarrow i + 1$
32:    end while
33: end while
34: Add missing edges to all nodes in $S$, with root $r$
35: Remove excess pooling nodes, beginning at root $r$
36: return $r$
```

We note that a simple breadth-first algorithm, in which new child nodes are generated for each current node, is only capable of producing trees. To enable generation of directed acyclic graphs, we allow the procedure that generates random nodes to return NULL (lines 21-22). The NULL value is used to indicate that rather than adding an edge to a new node, instead an edge should be added to reference an existing node, producing a directed acyclic graph. Because at this stage in the algorithm, the graph has not been fully constructed, we defer this process by adding the node to a multiset S for later processing (line 22). Hence the overall algorithm can be described as first constructing a tree, and then using the multiset S to add further edges to produce a directed acyclic graph. Otherwise, if a non-NULL randomly generated node is returned, we add the new edge, and also enqueue the new node.

After the main section of the algorithm, further processing is performed to add the 'missing' edges to the tree to produce a graph (line 34); this is detailed in Algorithm 3. In addition to this, postprocessing is required to validate the number of pooling nodes. Because each pooling node (either MAX or AVERAGE) reduces the spatial dimensions by $1/2$, there is a maximum number of pooling nodes before the image dimensions become 1×1, at which point further pooling is invalid. Because our pooling nodes use the stride and kernel size of 2, the maximum number of pooling nodes in a linear sequence can be calculated as the base-2 logarithm of the input image size, for a square image. In our implementation, we resolve this issue by counting the number of pooling nodes, and then replacing excess pooling nodes with a randomly generated convolutional node (line 35). The pooling nodes nearest to the input data are replaced first, as using pooling nodes before other operations (convolution, etc.) may lose important image data due to the dimensionality reduction.

D. Initialisation Algorithm Details

In this subsection, we detail the components that are used within the overall framework of the initialisation algorithm previously discussed. Algorithm 2 presents the process for determining the set R of restricted node types, used when generating new nodes to expand the graph.

Algorithm 2: Determine Disallowed Node Types
Input: A node n with level ℓ; a depth d; the boolean $nonleaf$

```
1: $R \leftarrow \emptyset$
2: $b \leftarrow$ True if an edge exists from $n$ to an INPUT node; else False
3: if $b$ or not $nonleaf$ then
4:     $R \leftarrow R \cup \{\text{INPUT}\}$
5: end if
6: if $\ell + 2 = d$ or $type(n) \in \{\text{SUM}, \text{CONCAT}\}$ then
7:     $R \leftarrow R \cup \{\text{BINARY}\}$
8: end if
9: return $R$
```

The inputs to Algorithm 2 are the current node n (to which the new nodes will be added) and its level ℓ, and the target depth d for the graph. The boolean value $nonleaf$ is computed in Algorithm 1 and stores whether a non-leaf node (i.e. a node of any type other than INPUT) has been added at the current level.

The algorithm first initialises R as an empty set (line 1). If there is already an edge from n to an INPUT node (condition 1), or if no non-leaf node has been added at the current level (condition 2), then INPUT is added as a restricted node type (lines 2-5). The justification for the first condition is that the nodes with arity $a > 1$ are SUM and CONCAT and it would be redundant to sum or concatenate the input image data with itself. The second condition is used to ensure that the depth of the directed acyclic graph is able to continue growing: if a non-leaf node has not yet been added at this level, adding a INPUT (i.e., leaf) node could cause the generation of the graph to terminate before reaching the target depth.

The second main step in Algorithm 2 determines whether nodes for binary operators (i.e., SUM and CONCAT) are permitted (lines 6-8). The first condition specifies that binary operators are only allowed if the 'parent' node has level $\ell < d - 2$. This condition is added because if the node has a level $\ell = d - 2$, a new SUM or CONCAT node would only be able to take its input from the image input pixels. If this occurred, this would be the same situation as previously discussed, in which redundant summation or concatenation of the image pixels would be produced. The second condition disallows chains of SUM and CONCAT nodes. That is, an edge from one of the two binary nodes to another binary node is not permitted. Note that unlike the previous cases discussed in Algorithm 2, this restriction is not strictly required. Rather, we include this condition as a constraint on the search space. If arbitrary sequences of binary operators are permitted, this can cause the number of branches generated by the algorithm to rapidly increase, potentially leading to overly complex and suboptimal CNNs. We note also that the CNN encoding itself is still general and, as it is designed to be used with evolutionary algorithms, genetic operators could be designed in a way that edges between nodes representing binary operators are permitted.

Algorithm 3 details the process used to add new edges based on the multiset S of 'missing' edges. That is, the algorithm adds edges specified in S so that the tree becomes a directed acyclic graph.

Algorithm 3: Add Missing Edges
Input: The multiset S containing each node p with a missing edge; the root node r of the directed acyclic graph

```
1: for $p$ in $S$ do
2:     $k \leftarrow 0$
3:     $t \leftarrow$ NULL
4:     $valid \leftarrow$ False
5:     while $k$ is less than some maximum iteration count do
6:         $t \leftarrow$ A random node in the graph with root $r$
7:         $b_1 \leftarrow$ True if there exists a path in the directed graph starting from $t$ and leading to $p$; else False
8:         $b_2 \leftarrow$ True if there is an edge from $p$ to $t$; else False
9:         if $type(t) \neq$ INPUT and not $b_1$ and not $b_2$ then
10:            $valid \leftarrow$ True
11:            break
12:        end if
13:        $k \leftarrow k + 1$
14:    end while
15:    if $valid$ then
16:        Add an edge from $p$ to $t$
17:    else
18:        Add an edge from $p$ to a new INPUT node
19:    end if
20: end for
```

For each node p in the set, the algorithm selects a random node n in the graph (line 6). As the node n may not be a valid choice (discussed in the following), the algorithm loops repeatedly until a valid node n is found, up to some maximum number of iterations. If this maximum is exceeded, an edge from p to an INPUT node is added. There are two reasons why adding an edge from node p to n may not be valid: one is that a cycle could be formed (line 7), or an edge from p to n may already exist (line 8). In addition, an INPUT node is not considered a valid choice for n (unless the maximum number of iterations is exceeded without identifying a valid n); this is because the goal of this method is to create additional connections between nodes internal to the graph (similar to the concept of skip connections, as discussed earlier).

Algorithm 4: Generate Random Node
Input: A set R of restricted node types

```
1:  $U \leftarrow \{\text{BINARY}, \text{CONV}, \text{POOL}, \text{INPUT}, \text{REFERENCE}\}$
2:  $P \leftarrow U - R$
3:  $t \leftarrow$ A random node type in $P$
4:  $n \leftarrow$ A new node
5:  if $t = $ BINARY then
6:      $q \leftarrow$ Uniformly generate a number between 0 and 1
7:      if $q < 0.5$ then
8:          $n.type \leftarrow$ SUM
9:      else
10:         $n.type \leftarrow$ CONCAT
11:     end if
12: else if $t = $ CONV then
13:     $n.type \leftarrow$ CONV
14:     $n.feature\_maps = $ Random choice in $\{64, 128, 256\}$
15: else if $t = $ POOL then
16:     $q \leftarrow$ Uniformly generate a number between 0 and 1
17:     if $q < 0.5$ then
18:         $n.type \leftarrow$ MAX
19:     else
20:         $n.type \leftarrow$ AVERAGE
21:     end if
22: else if $t = $ INPUT then
23:     $n.type \leftarrow$ INPUT
24: else if $t = $ REFERENCE then
25:     $n = $ NULL
26: end if
27: return $n$
```

Algorithm 4 details the method we use to randomly generate a node and its parameters; this method is based on that used in the CNN-GA paper [16], but we introduce the set R, which specifies node types that are disallowed. The set R is used to construct a set P of permitted node types (lines 1-2), from which a node type is randomly chosen with a uniform distribution (line 3). Note that the set U (line 1) contains different values than simply the available node types in the graph representation. That is, rather than including MAX and AVERAGE pooling nodes in U, we simply use a POOL type; similarly, there is a BINARY type, rather than SUM and CONCAT types. The reason for this is that each type in P is assigned equal probability, and we wish to choose uniformly between a convolutional node, a pooling node, or a binary operation. If, for example, MAX and AVERAGE pooling types were included separately in P, there would be a higher probability of generating a pooling node than a convolutional node. We would like to avoid this as pooling layers reduce the dimensionality and so can lose information if used excessively. Similarly, if there was a higher probability of a binary operation (SUM or CONCAT) than a convolutional node, this could result in excessively complex CNN architectures.

If the randomly selected node type is BINARY, a random number q is generated (with uniform distribution) between 0 and 1. If it is less than 0.5, a new SUM node is returned; otherwise, a CONCAT node is produced. If the selected node type is CONV, a new CONV node is generated, and the number of feature maps is chosen randomly. Specifically, the number of feature maps is chosen uniformly from $\{64, 128, 256\}$, which are the values used by the CNN-GA approach [16]. In addition, this fixed set of options for the number of feature maps contributes to reducing the search space (and hence improving the efficiency of search). If the randomly selected node type is POOL, a MAX pooling node is generated with probability 0.5; otherwise, an AVERAGE pooling node is used.

If the selected type is INPUT, the method simply returns a new INPUT node. The special REFERENCE type is used to implement the case where no new node is generated, but instead an edge to a existing node in the graph should be added. In this case, Algorithm 4 simply returns NULL. This value is then used as a flag by the initialisation method (Algorithm 1), which updates the set S of 'missing' edges to be processed, as was earlier discussed.

IV. Experiment Design

A. Overview

We test the proposed encoding and initialisation method on the CIFAR-10 benchmark data set [18]. CIFAR-10 consists of 10 classes and a total of 60,000 32×32 pixel images, of which 50,000 form the training set and 10,000 form the test set. During the architecture search process, we use only 10% of the full 50,000 training images, in order to improve the efficiency of the search. Specifically, during the architecture search, 4,500 images are used as the training set and 500 as the validation set. These subsets of the full 50,000 training images are selected using a stratified sampling process to maintain an even distribution of classes across the training and validation sets.

To demonstrate the effectiveness of the proposed encoding, we test its performance using a simple random search in which 200 CNN architectures are randomly generated using the proposed initialisation method. Each CNN is trained using the 4,500 images, and after each epoch, the performance is tested on the validation set. For each architecture, we record the maximum validation accuracy achieved. After training all architectures on the reduced data set, we select the three architectures with the best accuracy and re-train them using

TABLE I
THE RESULTS ACHIEVED BY RANDOM SEARCH WITH THE PROPOSED ENCODING IN COMPARISON WITH THE PEER COMPETITORS.

	Accuracy (%, CIFAR-10 Test Set)	**Parameters**	**GPU Days**
ResNet (depth=101) [6]	93.57	1.7 M	-
ResNet (depth=1,202) [6]	92.07	10.2 M	-
VGG [4]	93.34	20.04 M	-
DenseNet [7]	94.17	27.2 M	-
Hierarchical Evolution [14]	96.37	-	300
Block-QNN-S [11]	95.62	6.1 M	90
Genetic CNN [12]	92.90	-	17
Large-scale Evolution [13]	94.60	5.4 M	2,750
CGP-CNN [15]	94.02	1.68 M	27
CNN-GA [16]	95.22	2.9 M	35
Random search with the proposed encoding (trial #1, best accuracy)	92.45	1.45 M	1.33
Random search with the proposed encoding (trial #1, fewest parameters)	90.98	0.83 M	1.33
Random search with the proposed encoding (trial #2, best accuracy)	92.57	1.14 M	1.33
Random search with the proposed encoding (trial #2, fewest parameters)	91.44	0.75 M	1.33

45,000 training images and 5000 validation images. The reason for selecting three architectures (rather than the single best architecture) is to improve the likelihood of choosing an architecture that generalises from the reduced training set to the full training set. The epoch that gives the best validation accuracy is recorded for each architecture. Finally, each architecture is re-trained another time, up to the maximum epoch previously determined, using the full training set (50,000 images) *without* a validation set; the accuracy of the retrained CNN is then calculated on the test set.

B. Parameter Settings

To determine the performance of each CNN architecture, we use a training routine and learning rate schedule based on the settings used in [17]. That is, we use stochastic gradient descent [25] with a momentum of 0.9 and initial learning rate of 0.1 to train up to 200 epochs using a minibatch size of 128. The learning rate is decayed by a factor of 0.1 after the 60th, 120th and 160th epochs. We use the softmax cross-entropy loss as the loss function. In cases where an error occurred while training a CNN, such as the program exceeding available GPU memory, or a NaN (not-a-number) loss value being obtained, the CNN architecture was simply discarded.

To reduce overfitting, we apply weight decay with the coefficient 0.5×10^{-3}, which is the setting used in [17]. In addition, we employ a data augmentation technique based on the method used in [15] and [16]: a random horizontal flip is performed, and a random 32×32 crop is chosen after padding the image by 4 pixels on each side.

For the random search, it is necessary to specify a range of depths for the CNN population initialisation algorithm. In this implementation, we set the minimum depth to 6 and the maximum depth to 12. Although this limits the maximum depth during the random search, an evolutionary algorithm–based search could use genetic operators such as crossover and mutation to extend the representation to arbitrary depth; this will be investigated in future work.

V. Experiment Results

Table I shows the accuracy obtained on the CIFAR-10 test set (10,000 images) by the random search using our proposed encoding and by 10 state-of-the-art peer competitors, including both hand-crafted CNNs and CNNs produced by automatic design algorithms. All accuracies shown with the random search are those obtained on the test set after re-training the discovered architecture using the full 50,000 training images. For each model, the classification accuracy, the number of trainable parameters (in millions) and the GPU days required to determine the architecture are shown; this is a similar table format to that used in [16]. For the hand-crafted architectures, the 'GPU days' column is marked '-'.

For our proposed method, we show two separate trials of the random search. For each trial, we show the results from the CNN with the best accuracy and from the CNN with the fewest parameters. For the GPU days required, the architecture search itself takes about 10 hours using two GTX 1080 Ti GPUs; the re-training of the final three architectures requires approximately 12 hours with one GPU, such that the process takes approximately 1.33 GPU days overall.

Hence although the classification accuracies achieved by the random search are generally worse than those of the hand-crafted CNNs and other automatic design methods, this approach required significantly less computational resources, and provided promising performance despite the reduced data set during architecture search. Furthermore, the method was able to produce models with different complexities (numbers of parameters); for example, one generated CNN achieves 91.44% accuracy with only 750k parameters, after re-training using 50,000 images.

Fig. 2 shows the graph representation for the best architecture from trial #2 of the random search. In the diagram, 'MP' and 'AP' denote max and average pooling nodes, respectively; the 'Conv' nodes represent convolutional building blocks, where the value in parentheses is the number of feature maps.

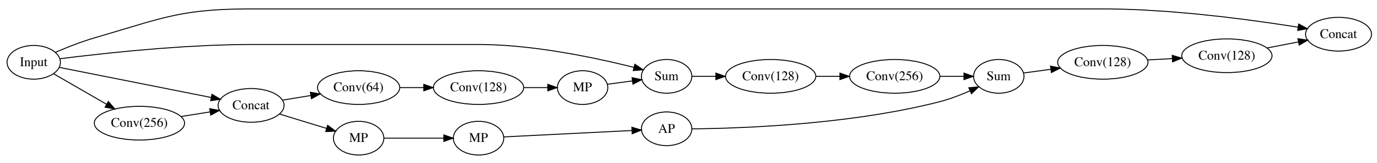

Fig. 2. The best CNN architecture generated by the random search (trial #2), with an accuracy of 92.57% after re-training on the full 50,000 training images.

VI. Conclusions

In this paper, we presented an encoding strategy for CNN architectures based on a directed acyclic graph representation, designed to be used for evolutionary deep learning. In contrast to encoding methods employed by many evolutionary computation–based architecture search approaches in the literature, our proposed representation imposes fewer restrictions, thus allowing more architectures to be represented. We believe that this increases the potential for automatically determining an architecture that is well suited to the target data set. Furthermore, we introduced a method for random initialisation of this encoding, which could be used for the population initialisation step in an evolutionary algorithm. We demonstrated the potential of this approach with a random search using only 10% of the CIFAR-10 training data during the architecture search, but which still achieved promising results after the three best discovered architectures were re-trained on the full 50,000 training images. In particular, the method shows good capability for generating models with a variety of complexities. Due to the reduced training set used during the random search, this method also required much less computation time compared to other methods. Given the promising results despite the limitations (i.e., a simplistic search algorithm and a small data set during search), it appears that the proposed CNN encoding has good potential to gain improved performance if used with a more sophisticated architecture search approach. In future work, we intend to implement an evolutionary deep learning method based on this encoding to further improve the accuracy.

References

[1] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems*, pp. 1097–1105, 2012.

[3] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, Nov 1998.

[4] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *Proceedings of the 32nd International Conference on Machine Learning*, (Lille, France), 2015.

[5] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, June 2015.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, June 2016.

[7] G. Huang, Z. Liu, L. v. d. Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2261–2269, July 2017.

[8] Y. Sun, B. Xue, and M. Zhang, "Evolving deep convolutional neural networks for image classification," *ArXiv e-prints*, p. arXiv:1710.10741, Oct. 2017.

[9] B. Zoph and Q. V. Le, "Neural Architecture Search with Reinforcement Learning," in *Proceedings of the 2017 International Conference on Learning Representations*, (Toulon, France), 2016.

[10] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing Neural Network Architectures using Reinforcement Learning," in *Proceedings of the 2017 International Conference on Learning Representations*, (Toulon, France), 2016.

[11] Z. Zhong, J. Yan, and C.-L. Liu, "Practical network blocks design with q-learning," in *Proceedings of the 2018 AAAI Conference on Artificial Intelligence*, (Louisiana, USA), 2018.

[12] L. Xie and A. L. Yuille, "Genetic CNN," in *Proceedings of 2017 IEEE International Conference on Computer Vision*, (Venice, Italy), pp. 1388–1397, 2017.

[13] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. Le, and A. Kurakin, "Large-Scale Evolution of Image Classifiers," in *Proceedings of Machine Learning Research*, (Sydney, Australia), pp. 2902–2911, 2017.

[14] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, "Hierarchical representations for efficient architecture search," in *Proceedings of 2018 Machine Learning Research*, (Stockholm, Sweden), 2018.

[15] M. Suganuma, S. Shirakawa, and T. Nagao, "A genetic programming approach to designing convolutional neural network architectures," in *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, (Berlin, Germany), pp. 497–504, ACM, 2017.

[16] Y. Sun, B. Xue, M. Zhang, and G. G. Yen, "Automatically Designing CNN Architectures Using Genetic Algorithm for Image Classification," *ArXiv e-prints*, p. arXiv:1808.03818, Aug. 2018.

[17] Y. Zhu, Y. Yao, Z. Wu, Y. Chen, G. Li, H. Hu, and Y. Xu, "GP-CNAS: Convolutional Neural Network Architecture Search with Genetic Programming," *arXiv e-prints*, p. arXiv:1812.07611, Nov. 2018.

[18] A. Krizhevsky and G. E. Hinton, "Learning multiple layers of features from tiny images," 2009. *Online: https://www.cs.toronto.edu/~kriz/cifar.html*.

[19] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[20] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–1780, Nov. 1997.

[21] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Training Very Deep Networks," in *Advances in Neural Information Processing Systems*, (Montreal, Canada), 2015.

[22] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, 2011.

[23] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning*, ICML'15, pp. 448–456, 2015.

[24] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of CVPR*, pp. 2818–2826, 2016.

[25] L. Bottou, "Stochastic gradient descent tricks," in *Neural Networks: Tricks of the Trade: Second Edition*, pp. 421–436, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.