Image Classification with Evolved Convolutional Neural Networks

Image Classification with Evolved Convolutional Neural Networks

Philip W. Darnowsky

A Thesis in the Field of Software Engineering

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

March 2022

Abstract

Convolutional neural networks (CNNs) are a well-established technique for image classification problems. While the topology of a CNN strongly affects the performance of that CNN, designing a CNN's topology remains a difficult task, often with nothing better than some empirical rules-of-thumb for guidance. Evolutionary algorithms are a family of metaheuristics that can be applied to optimization problems where good solutions are hard to create from first principles, but the quality of a given solution is easy to measure. In this research, we develop and evaluate several variations on an algorithm, SDAG, which applies evolutionary methods to finding performant topologies for CNN-based image classifiers.

Acknowledgements

I would not have gotten nearly this far without the aid and support and good cheer of my research advisors, Dr. Hongming Wang and Dr. Sylvain Jaume. Credit is also due to Nada El-Newahy, Gail Dourian, and the staff of the Harvard libraries for their invariably swift and efficient help.

To my friends and colleagues at the MBTA's Customer Technology Department, who supported me in many ways ranging from helping to track down data, to letting me sound out ideas on them, to simple encouragement: it has been the greatest privilege of my career so far to work with you on the grand project of moving millions of people around Massachusetts (and parts of Rhode Island). In particular, I would like to recognise Dave Barker, Gabe Durazo, David Gerstle, Hayley Garment, Cora Grant, Akira Hakuta, Cristen Jones, Maeg Keane, Beatrix Klebe, Tyler Lavoie, Crissy Leach, Ryan Mahoney, Adamo Maisano, Eddie Maldonado, Ashli Molina, Theo Molina, Logan Nash, Heather Purcell, Jessie Richards, Molly Rock, Sky Rose, Dan Salomon, Karl Stone, and Paul Swartz. Apologies to anyone I left out–we've had too many adventures for me to keep track of them all.

I would like to thank everyone who helped me maintain my physical and mental health during this process, especially given that it coincided with a pandemic: Dr. Robert Atkind and his staff; Evan Gomez; Fred Pagano; Joyce Shields and company; and Dr. Steven Varga.

Thanks to Sadie–she knows what she did.

Last and most, to Peggy, for everything.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter I. Introduction

Since the late 1980s–more than 30 years ago as of this writing–researchers have known that sufficiently complex artificial neural networks (ANNs) "can approximate virtually any function of interest to any desired degree of accuracy" (Hornik et al., 1989). Moreover, there are practical methods (most commonly gradient descent by backpropagation (Rumelhart et al., 1986) (Lecun et al., 1998)), given an ANN and a "function of interest", to adjust the weights of that ANN to better approximate the function (Aggarwal, 2018). A particular subset of ANNs, the convolutional neural networks (CNNs), have proven effectiveness in many domains, but especially in their original application of image classification (Rawat & Wang, 2017), and can be "trained" by the same methods as any other ANNs.

By the early-to-mid 2000s, graphics processing units (GPUs) that could be programmed through a specialized assembly language were available on the mass market, and researchers showed that they had great potential for high-performance massively-parallel numerical computation in general (Bolz et al., 2003) (Krüger & Westermann, 2003) (Steinkraus et al., 2005), and training of ANNs, including CNNs in particular (Steinkraus et al., 2005) (Chellapilla et al., 2006). Since then, GPUs have become the predominant platform for working with CNNs and other machine learning models (Nguyen et al., 2019).

We have, then, proven techniques for automatically training the weights of ANNs, and relatively inexpensive hardware coprocessors to greatly accelerate these

techniques. For all of this progress, however, one important aspect of creating an ANN remains arguably more art than science: an ANN can be thought of as a graph of nodes called "neurons", and while it is well established that the topology of this graph can greatly influence both the efficiency and the performance of an ANN (Solla, 1992), there are few hard-and-fast rules about how to choose a topology. This task still usually calls for a great deal of human expertise, effort, and trial-and-error (Jaafra et al., 2019).

The problem of finding an ANN topology for a given application is a problem where it's difficult to design a good solution from first principles, as it's largely unclear what those principles even are. However, given a proposed solution, it's simple (though perhaps computationally very expensive) to evaluate the quality of that solution: we construct an ANN with the given topology, train it on a training dataset, then evaluate its performance on a separate test dataset (Aggarwal, 2018). When we have a problem that's difficult to solve, but easy to evaluate solutions for, there is a family of robust and versatile heuristics that we can consider using. That family is the family of evolutionary algorithms (EAs).

If ANNs are a "biomimetic" method, based on a model of physiological processes in a living animal, we could call EAs "ecomimetic", as they simulate some of the dynamics present in real ecosystems. According to the theory of evolution, given certain conditions that are found in natural ecosystems, with the passage of time we would expect the typical organism living in that ecosystem to become better adapted to its ecological niche (Holland, 1992). In a typical EA, the "organisms" are proposed solutions to some problem, and we have a way of measuring the quality, or "fitness", of a solution. We begin with a "population" of solutions that are randomly generated, and hence probably not very good. Given such a population, a single iteration of an EA "breeds" a new population, the contents of which are generated by combining

and altering solutions present in the current population. This breeding process is set up so that solutions with a higher fitness will tend to pass on their characteristics to the next generation more often than less fit solutions. Over the course of many generations, we expect the average fitness of a solution to rise (Fogel, 1994).

Part of the appeal of EAs is that they can be applied to problems that are difficult-to-impossible to solve in a closed form; as Lozano puts it, they "can deal with multimodalities, discontinuities and constraints, with noisy functions, multiple-criteria decision making processes or with problems given by a simulation model" (Lozano, 2002). The idea of using EAs to find an effective topology for an ANN also goes back to the 1980s (Kampfner & Conrad, 1983) (Miller et al., 1989). However, even though CNNs also began to be investigated during that decade (Fukushima, 1980) (LeCun et al., 1989), there seems to be no published research on the topic of evolving CNN topologies until 2017 (Suganuma et al., 2017) (Real et al., 2017) (Xie & Yuille, 2017), which was less than five years ago as we write this, and it was only in 2020 that methods without significant restrictions on the topologies of the resulting CNNs were first introduced by Yuan et al. (Yuan et al., 2020). Their experimental results in that work are very promising, and their methods appealingly flexible. In this work, we hope to contribute some fundamental knowledge to this new and exciting line of research.

Chapter II. Background

2.1. Artificial Neural Networks

Artificial neural networks (ANNs) are a family of computational methods based on a simplified mathematical model of the structure of biological brains (Aggarwal, 2018). The fundamental unit of the ANN is the "neuron" (Aggarwal, 2018). Formally, we can define a neuron as a function $n : \mathbb{R}^k \to \mathbb{R}$, where:

$$n(\vec{x}) = \varphi(b + \vec{x} \cdot \vec{w}) \tag{2.1}$$

$\vec{x}$ is a k-dimensional vector, $\vec{w}$ is a constant k-dimensional "weight" vector, b is a constant scalar "bias" value, $\varphi : \mathbb{R} \to \mathbb{R}$ is a (typically nonlinear) "activation function", and $(\cdot) : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}$ is the vector dot-product operator $(\cdot)(\vec{x}, \vec{y}) = \sum_{i=0}^{k-1} x_i \; y_i$.

Individual neurons are arranged into a directed graph to form an ANN. If this graph is acyclic, we call this a "feedforward" neural network. If any cycles exist, the network is "recurrent" (Xin Yao, 1999). In this paper, we will consider only feedforward networks, and hence only directed acyclic graphs (DAGs), except where noted. Certain neurons $\{o_0, ..., o_{b-1}\}$ in the ANN are designated as outputs from the ANN. There are also nodes $\{i_0, ..., i_{a-1}\}$ in an ANN that represent not neurons, but rather the inputs to the ANN. No edges in the graph terminate at an input node, and

for any neuron in the graph, there is a path (perhaps with multiple edges) to that neuron from at least one input node. There is a one-to-one correspondence between edges coming into a neuron, and the elements of the weight vector of that neuron. That is, if we let $N_n(\vec{x})$ represent the output of neuron n in an ANN N when vector $\vec{x}$ is input to N, we can rewrite (2.1) like so:

$$N_n(\vec{x}) = \varphi(b + \sum_{e \in E} w_e v_e(\vec{x})) \tag{2.2}$$

Here, φ and b are defined as in (2.1); E is the set of all edges in N terminating at n; $w_e \in \mathbb{R}$ is the weight associated with edge e; and, if n' is the input node or neuron from which edge e originates, $v_e : \mathbb{R}^a \to \mathbb{R}$ is defined:

$$v_e(\vec{x}) = \begin{cases} \vec{x}_k & \text{if } n' \text{ is input node } i_k \\ N_{n'}(\vec{x}) & \text{otherwise} \end{cases} \tag{2.3}$$

Putting (2.2) and (2.3) together, we have a method by which we can assign input values to the input nodes of an ANN, then recursively calculate the values of all neurons. In this way we can consider an ANN with a input nodes and b output nodes to define a function $N : \mathbb{R}^a \to \mathbb{R}^b$, where $N(\vec{x}) = \langle N_{o_0}(\vec{x}), ..., N_{o_{b-1}}(\vec{x}) \rangle$ (Aggarwal, 2018).

Optimization of the weights of an ANN N to approximate a target function $f : \mathbb{R}^a \to \mathbb{R}^b$ is generally done by supervised learning on a training dataset $D \subset \mathbb{R}^a$ using gradient descent via backpropagation (Lecun et al., 1998) . Taking that statement one piece at a time, "supervised learning" means that we already know the value of $f(\vec{x})$ for each element $\vec{x} \in D$, and we will repeatedly present N with an element $\vec{x} \in D$, calculating a predicted value $N(\vec{x})$ (Lecun et al., 1998). "Gradient descent" means, for each $\vec{x}$ we present to N, we calculate $L(f(\vec{x}), N(\vec{x}))$ for some "loss function"

$L : (\mathbb{R}^b, \mathbb{R}^b) \to \mathbb{R}$ that measures the degree of error in the predicted value $N(\vec{x})$ when compared to the known value $f(\vec{x})$ (Lecun et al., 1998) (Aggarwal, 2018). For each weight w of N, we then find the partial derivative $\frac{\partial L}{\partial w}$, and we calculate an updated weight $w' = w - \epsilon \frac{\partial L}{\partial w}$, with the "learning rate" ϵ a small positive value (Lecun et al., 1998). "Backpropagation" is the name of the dynamic programming algorithm that allows us to efficiently find those partial derivatives (Aggarwal, 2018). When each $\vec{x} \in D$ has been so used to adjust the weights of N once, we say that we have trained N for one "epoch" (Aggarwal, 2018). While it is generally necessary to train an ANN for multiple epochs to achieve an acceptable level of performance, training an ANN for too many epochs introduces the risk of "overfitting". Overfitting is characterized by poor generalization; that is, an overfit ANN's performance on the training data may be very good, but its performance on any other dataset will generally be significantly worse (Aggarwal, 2018).

2.2. Convolutional Neural Networks

Convolutional neural networks (CNNs) are a class of ANNs originally developed in the 1980s for visual pattern recognition tasks, modeled on the structure of the vertebrate visual cortex (Hubel & Wiesel, 1962) (Fukushima, 1980) (LeCun et al., 1989). In this simplified model of vision, we can think of the input, i.e. the light that enters an animal's eye and falls upon the retina, as a two-dimensional matrix of pixels, just as we might represent an image in a computer program.

The first element to process this input is a grid of "feature detectors", each associated with a set of coordinates that specify a particular element of the input. For each detector, the region of some given size around that point is the "receptive field" of the detector (Hubel & Wiesel, 1962) (Aggarwal, 2018). Each detector analyses the subset of the input that falls into its receptive field, looking for the presence or

absence of a set of simple geometrical features, which set is common to all detectors in the layer (Aggarwal, 2018). The first layer thus assembles a "feature map" for each detectable feature: for any given point in a feature map in the output of the first layer, the value at that point indicates the degree to which the feature detector whose receptive field centers on that point considers its input to resemble the feature in question (Aggarwal, 2018).

The output from this first layer of detectors is fed into another layer of detectors. Just as the first layer, each detector in the second layer is associated with a certain receptive field, this time consisting of the subset of first-layer detectors within a given region. Also like the first-level detectors, the second-level detectors assemble their input into feature maps. However, the input to the second layer is the set of feature maps output by the first layer, and rather than detecting simple features from raw input, the second layer detects slightly more complicated features formed by combinations of simple features (Aggarwal, 2018). More layers may follow, with each layer forming more and more complex features from the simpler features detected by the layer below, until features of great complexity can be recognized (Fukushima, 1980).

For simplicity, we will assume in this paper, unless stated otherwise, that the input to a CNN is always a rank-3 tensor representing a two-dimensional image in RGB color (Schwarz et al., 1987), written in the form $T = \{t_{cxy} : 0 \leq c < 3, 0 \leq x < h, 0 \leq y < w\}$, for some positive height and width in pixels h and w. We call the subset $\{t_{ixy} : 0 \leq x < h, 0 \leq y < w\}$ the "i^{th} plane" of T. As a consequence of this assumption, intermediate results within a CNN will also be rank-3 tensors, though generally of a different size than the input.

Although we concentrate on rank-3 tensor input in this work, the techniques in this section can be extended to input of other ranks. One-dimensional CNNs, that

is those that take a rank-1 tensor (a.k.a. a vector) as input, have been applied to detection of defects in machinery, in which process they act upon time series data represented as vectors (Ince et al., 2016). CNNs are also used to evaluate video input by representing it as a rank-4 tensor, adding a time dimension to the spatial and color dimensions of still images (Baccouche et al., 2011) (Ji et al., 2013).

As a general ANN can be represented by a DAG of neurons, a CNN can be represented by a DAG of units called "layers" (Aggarwal, 2018). Unlike the arbitrary number of inputs we can attach to a neuron, a layer in a CNN has a defined arity, i.e., a specific number of inputs that it must have (Irwin-Harris et al., 2019). As long as that condition is satisfied, the architecture of a CNN may otherwise be an arbitrary DAG. Apart from the binary layers that we will discuss in Section 2.2.5, which have an arity of 2, all of the following layer types have arity 1.

2.2.1 Convolution Layers

The convolution operation which gives CNNs their name is a generalization of the dot product from vectors, i.e. tensors of rank 1, to tensors of arbitrary rank. A convolution layer defines a function $f : \mathbb{R}^a \to \mathbb{R}^b$ in terms of a set $K = \{K_0, ..., K_{b-1}\}$ of tensors called "kernels" or "filters" (Aggarwal, 2018). For some given odd positive integers h' and w', with $h' \ll h$ and $w' \ll w$ (and often with $h' = w'$), each kernel in K has dimensions $a \times h' \times w'$.

A convolution layer uses each kernel k_i in turn to generate the i^{th} plane of its output. Given an input tensor $T \in \mathbb{R}^{a \times h \times w}$, and a kernel $k_i \in \mathbb{R}^{a \times h' \times w'}$, informally speaking we can think of positioning k_i so that it overlaps with some subset $T' \subset T$, with $T' \in \mathbb{R}^{a \times h' \times w'}$ just as k_i. Note that, since h' and w' are both odd, there is some x and y such that the center of T' is the set $\{t_{nxy} : t_{nxy} \in T, 0 \leq n < c\}$. The value of the element of $f(T)$ with coordinates (i, x, y) is then the convolution of k_i with T',

written $k_i \star T'$, and defined as the sum of the products of each element of k_i with the corresponding element of T'. Alert readers might now be wondering what happens if x and y are such that some of T' would be outside the bounds of T. There are several ways to deal with this (Aggarwal, 2018); we will take the approach of pretending that those missing coefficients from T are in fact present, but equal to 0, so that they end up not affecting the final sum. As in a general ANN, we can also add a fixed bias term to each output value; each kernel may have its own bias value. Algorithm 2.1 on page 10 describes the operation of a convolution layer more formally.

In addition to its kernels, a convolution layer does have other parameters such as "stride" and "dilation". We will not touch on dilation here nor use it in our experiments; the curious are referred to Yu and Koltun (Yu & Koltun, 2016). Stride is described in Section 2.2.4. We can consider Algorithm 2.1 to implicitly use a stride of 1 and a dilation of 0.

2.2.2 Depthwise Separable Convolution Layers

A depthwise separable convolution consists of "a depthwise convolution, i.e. a spatial convolution performed independently over each channel of an input, followed by a pointwise convolution, i.e. a 1x1 convolution, projecting the channels output by the depthwise convolution onto a new channel space" (Chollet, 2017).

In other words, given a tensor $T \in \mathbb{R}^{c \times h \times w}$, let P_i be the i^{th} plane of T. We treat each P_i as a separate tensor, with $P_i \in \mathbb{R}^{1 \times h \times w}$, and for each P_i we define a corresponding convolution kernel, $k_i \in \mathbb{R}^{1 \times h' \times w'}$. For each P_i, we then calculate the "spatial convolution" $P'_i = k_i \star P_i$. Now we reassemble the various P'_i, defining a tensor $T' \in \mathbb{R}^{c \times h \times w}$ such that the i^{th} plane of T' is equal to P'_i.

Let c' be the desired number of channels in the output, and define c' more convolution kernels, $\{k'_0, ..., k'_{c'-1}\}$, with each $k'_i \in \mathbb{R}^{c \times 1 \times 1}$. The result of the depthwise

Algorithm 2.1 Applying a convolution layer to a tensor

```
1: function CONVOLUTION(T, kernels)
2:     half_h' ← (kernels[0].height − 1)/2
3:     half_w' ← (kernels[0].width − 1)/2
4:     output ← tensor of size kernels.length × T.height × T.width, all elements
   initialized to 0
5:     for i' ← 0, kernels.length − 1 do
6:         kernel ← kernels[i']
7:         for x ← 0, h − 1 do
8:             for y ← 0, w − 1 do
9:                 output_value ← kernel.bias
10:                for Δ_x ← −half_h', half_h' do
11:                    for Δ_y ← −half_w', half_w' do
12:                        if 0 ≤ x + Δ_x < h and 0 ≤ y + Δ_y < w then
13:                            x' = Δ_x + half_h'
14:                            y' = Δ_y + half_w'
15:                            for i ← 0, T.depth − 1 do
16:                                kernel_value ← kernel[i][x'][y']
17:                                input_value ← T[i][x + Δ_x][y + Δ_y]
18:                                output_value ← output_value+
   kernel_value ∗ input_value
19:                            end for
20:                        end if
21:                    end for
22:                end for
23:                output[i'][x][y] ← output_value
24:            end for
25:        end for
26:    end for
27:    return output
28: end function
```

separable convolution is then $T'' \in \mathbb{R}^{c' \times h \times w}$, where the i^{th} plane of T'' is equal to $k'_i \star T'$. Algorithm 2.2 on page 11 demonstrates depthwise separable convolution.

Algorithm 2.2 Applying a depthwise separable convolution layer to a tensor

```
1: function DEPTHWISE_SEPARABLE_CONVOLUTION(T, depthwise_kernels,
   pointwise_kernels)
2:     T' ← empty tensor same size as T
3:     for i ← 0, T.depth − 1 do
4:         T'[i] ←CONVOLUTION(T[i], [depthwise_kernels[i]])
5:     end for
6:     return CONVOLUTION(T', pointwise_kernels)
7: end function
```

2.2.3 Activation Layers

Just as the final step in evaluating a neuron in a general ANN is to apply an activation function, convolution and depthwise separable convolution layers are typically followed by an activation layer. In this work, we will be using "rectified linear unit" (Aggarwal, 2018) activation, "ReLU" for short. The ReLU function, $\varphi : \mathbb{R} \to \mathbb{R}$, is defined $\varphi(x) = \max(0, x)$. All a ReLU layer does, then, is apply the ReLU function to every value in the input tensor. Formally, a ReLU layer implements a function $\Phi : \mathbb{R}^{c \times h \times w} \to \mathbb{R}^{c \times h \times w}$, and if tensor $T = \{t_{ixy} : 0 \leq i < c, 0 \leq x < h, 0 \leq y < w\}$, then $\Phi(T) = \{t'_{ixy} : t'_{ixy} = \max(0, t_{ixy}), t_{ixy} \in T\}$ (Aggarwal, 2018).

2.2.4 Pooling Layers

Pooling layers, informally, operate on every plane of their input separately, reducing small square regions of each plane to a single value (Aggarwal, 2018). Two common types of pooling, both of which we will use in our experiments, are "max-pooling" and "average-pooling". Max-pooling maps each region of its input to the maximum value found in that region. Average-pooling maps each region to the

mean of values found in that region. Note that, in contrast to a convolution layer, the output of a pooling layer always has the same depth as the input (Aggarwal, 2018).

In addition to the pooling operation and the region size, a pooling layer is defined by a parameter called "stride" (Aggarwal, 2018). Intuitively, we can think of the stride as how far the region "moves" at each step. In this work, we will be exclusively using pooling layers with a 2×2 input region and a stride of 2. We will not examine the interaction of stride and region size in detail here; suffice to say that, with the parameters we are using, if we apply pooling to a tensor in $\mathbb{R}^{c \times x \times y}$, the output will be in $\mathbb{R}^{c \times (x/2) \times (y/2)}$ (Aggarwal, 2018).

Algorithm 2.3 on page 12 shows how max-pooling is applied to an input tensor; as described above, it uses a region size of 2×2 and a stride of 2. We will also assume, for simplicity, that the height and width of the input are divisible by 2. The algorithm for average-pooling is the same, except that on line 10 we calculate the mean, rather than the maximum, of the region in question.

Algorithm 2.3 Max-pooling a tensor

```
1:  function MAXPOOL(T)
2:      output_height ← T.height/2
3:      output_width ← T.width/2
4:      output ← empty tensor of size (T.depth, output_height, output_width)
5:      for i ← 0, T.depth − 1 do
6:          for j ← 0, output_height − 1 do
7:              for k ← 0, output_width − 1 do
8:                  upper_left ← (i, j ∗ 2, k ∗ 2)
9:                  region ← 1 × 2 × 2 slice of T with upper-left corner at upper_left
10:                 output[i][j][k] ← largest value in region
11:             end for
12:         end for
13:     end for
14:     return output
15: end function
```

2.2.5 Binary Layers

We can also define layers that implement binary operations on tensors. Two kinds that we will use are "sum layers" and "concatenation layers". Sum layers perform element-wise addition on two tensors of the same size. Concatenation layers return one tensor "stacked" on another of the same height and width. That is, if $T \in \mathbb{R}^{c \times h \times w}$, and $T' \in \mathbb{R}^{c' \times h \times w}$, the concatenation of them, written $T \oplus T'$, is such that $T \oplus T' \in \mathbb{R}^{(c+c') \times h \times w}$. The first c planes of $T \oplus T'$ are equal to the planes of T, and the remaining c' planes of $T \oplus T'$ are equal to the planes of T'. We can (and, in Chapter 3, do) extend both operations to tensors of incompatible shapes by adopting the convention that each matrix is padded with 0s as necessary to make the sizes compatible. The details of how we do this padding are shown in Algorithm 2.4 on page 14.

2.3. Evolutionary Algorithms

Evolution is one of the pillars of the modern understanding of biology. Some of the main ideas of the theory of evolution include the following (Bowler, 2003):

- Individuals produce offspring, either alone (asexual reproduction) or with the cooperation of another individual (sexual reproduction).
- The observable characteristics of an individual (the phenotype) are an expression of a collection of genetic information (the genotype).
- Genetic information is copied from parent(s) to children. In the case of sexual reproduction, crossover occurs, resulting in children with genotypes consisting of a mixture of both parents' genotypes.

Algorithm 2.4 Matching shapes of tensors to match sizes for sum or concatenation

function MATCH_SHAPES($Ts, match\_depth?$)
    if $match\_depth?$ **then**
        $max\_depth \leftarrow$ largest depth of all tensors in Ts
    end if
    $max\_height \leftarrow$ largest height of all tensors in Ts
    $max\_width \leftarrow$ largest width of all tensors in Ts
    $output \leftarrow [\,]$
    for all $T \in Ts$ **do**
        if $match\_depth?$ **then**
            $depth\_difference \leftarrow max\_depth - T.depth$
        else
            $depth\_difference \leftarrow 0$
        end if
        $height\_difference \leftarrow max\_height - T.height$
        $width\_difference \leftarrow max\_width - T.width$
        $depth_padding_front \leftarrow$FLOOR$(depth_difference/2)$
        $depth\_padding\_back \leftarrow depth\_difference - depth\_padding\_front$
        $height_padding_top \leftarrow$FLOOR$(height_difference/2)$
        $height\_padding\_bottom \leftarrow depth\_difference - height\_padding\_top$
        $width_padding_left \leftarrow$FLOOR$(width_difference/2)$
        $width\_padding\_right \leftarrow width\_difference - width\_padding\_left$
        $padded\_tensor \leftarrow T$ with 0s to pad on each side per the padding values found in previous six lines
        $output$.APPEND$(padded_tensor)$
    end for
    return $output$
end function

- The process of reproduction may be perturbed by randomly occurring mutations, by which mechanism novel genes can be introduced.

- Individuals exist in an environment, and not all individuals are equally well suited for existence in this environment. Due to the peculiarities of their respective phenotypes and the nature of their particular environment, some individuals will be "fitter" than others. Fitter individuals, by definition, tend to have more offspring than less fit individuals. We say that there has been "selection for" the traits of fitter individuals, and "selection against" those of less fit.

- As a result of the conditions above, over repeated cycles of reproduction and selection, the overall fitness of individuals in the population will tend to rise.

None of the above phenomena are restricted to the biological world (Holland, 1992). An individual might be a biological organism, or it might be a data structure in a computer's memory. Genes may be encoded in DNA, or in bits. Phenotypes may result from the transcription of DNA into proteins, or from applying a function to a value. By the mid-1960s, research into several varieties of evolutionary algorithms—optimization metaheuristics inspired by natural evolution—was well established (De Jong et al., 1997).

A generic evolutionary algorithm might have this general structure (Xin Yao, 1999) (Ashlock, 2006) (Saxena & Saad, 2006):

1. Generate the first generation of individuals.

2. Evaluate all members of the current generation against a given fitness function. A fitness function is a problem-specific function that measures how good a solution a given individual is to the problem in question. Depending on our

formulation of the problem, we can adopt either the convention that a higher fitness value indicates better performance, or vice versa. For example, if our goal was to evolve a binary classifier, we might use the F_β score (van Rijsbergen, 1979) (Chinchor, 1992) of that classifier on a validation set, with a higher score indicating a fitter individual. Contrariwise, if evolving a multiclass classifier, the cross-entropy loss (Aggarwal, 2018) (Bridle, 1990) might be a useful fitness function, with a lower value indicating higher fitness.

3. If the given termination conditions are met, terminate and return the current population as the result of the algorithm. Termination conditions might include the reaching of a given maximum number of generations or amount of wall-clock time; the generation of an individual more fit than some threshold; or stagnation, defined as no improvement in fitness over some number of generations.

4. If the termination conditions are not met, breed a new generation of individuals from the current generation. This step is also dependent on the problem and the formulation of the problem, but typically analogues of the natural-world phenomena of mutation, asexual reproduction, and/or sexual reproduction with genetic crossover are used. The method of reproduction should have the property that fitter individuals will tend to reproduce more than less fit individuals.

5. Replace the current generation with the newly-bred generation and repeat from step 2.

2.3.1 Direct vs. Indirect Representations

In every evolutionary algorithm, the questions arise of how to represent a genotype, and how to map a genotype to a phenotype (Granadeiro et al., 2013) (Rothlauf, 2006). Let the function f map genotypes to phenotypes for some evolutionary

algorithm. If f is the identity function $f(g) = g$, we are using a "direct representation" (Rothlauf, 2006). Otherwise, we are using an "indirect representation". Per Granadeiro et al., the "two main requirements for a representation are encoding all the possible solutions of the problem and enabling the application of the variation operators to them (crossover and mutation)" (Granadeiro et al., 2013). Indirect representation arises because, for a given problem, there may be no known representation suitable for both "application of the variation operators" and the fitness evaluation central to EAs. In this project, we will be using an indirect representation. In fact, as we will see in Chapter 3, not only is our representation indirect, but the function f that we use to turn a genotype into a phenotype is stochastic, due to our use of the stochastic He normal initializer (He et al., 2015).

2.4. Related Work

In 1983, Kampfner and Conrad combined ANNs of one or a small number of neurons with an evolutionary algorithm. They were not interested in producing ANNs as an end in itself, but rather as a means to simulating the "evolutionary selection circuits model of learning" (Kampfner & Conrad, 1983) in biological brains. Nevertheless, in the process, they evolved ANNs which were able to solve simple problems in classification and control (Kampfner & Conrad, 1983). The first published work to focus on evolving ANNs *per se* may be that of Miller et al. in 1989 (Miller et al., 1989), which cites earlier unpublished work. Miller et al. used a genetic algorithm to produce feedforward ANN topologies and backpropagation to train ANN weights (Miller et al., 1989); in a broad sense, that is the same approach we use in this paper. A 1990 paper by Fahlman evolved recurrent ANNs (ANNs with cycles), but was limited in that it could only produce recurrent connections from one neuron to itself (Fahlman, 1990). By contrast, Angeline et al.'s GNARL algorithm, introduced

in 1994, could evolve arbitrary recurrent ANN topologies (Angeline et al., 1994).

In 2002, Stanley and Miikkulainen published an influential paper that introduced NEAT: the Neuroevolution of Augmenting Topologies method (Stanley & Miikkulainen, 2002). NEAT uses a direct encoding of an ANN as a graph, in which each genome contains a single gene corresponding to each neuron and edge of the graph (Stanley & Miikkulainen, 2002). An edge gene also contains that edge's weight, and the evolutionary methods of NEAT operate on both the topology and weights of the ANN (Stanley & Miikkulainen, 2002). NEAT was not the first algorithm to use evolutionary methods (as opposed to e.g. backpropagation) to optimize weights as well as topology; it was preceded by work such as that of Fullmer and Miikkulainen and that of Dasgupta and McGregor (Fullmer & Miikkulainen, 1991) (Dasgupta & McGregor, 1992). However, NEAT addressed several perceived shortcomings in earlier algorithms, and Stanley and Miikkulainen were able to show that NEAT was a viable method to evolve ANNs for control problems (Stanley & Miikkulainen, 2002). In the following years and through to recent times, numerous variations and refinements on NEAT have been invented (Whiteson et al., 2005) (Chen & Alahakoon, 2006) (Pardoe et al., 2005) (Miguel et al., 2008) (Kassahun & Sommer, 2005) (Siebel & Sommer, 2008) (Tan et al., 2009) (Kohl & Miikkulainen, 2012) (Wang et al., 2013) (Caamaño et al., 2015). One such variation is Stanley et al.'s HyperNEAT algorithm (Stanley et al., 2009), which, while not producing CNNs, is based on a similar insight, to wit, that "[b]iological neural networks rely on exploiting [geometric] regularities for many of their functions. For example, neurons in the visual cortex are arranged in the same retinotopic two-dimensional pattern as photoreceptors in the retina. That way, they can exploit locality by connecting to adjacent neurons with simple, repeating motifs" (D'Ambrosio et al., 2014).

The earliest reference we were able to find to applying EAs to CNNs specifically

was not until 2014, in the work of Koutník et al.; those authors do claim this to be the "first use of deep learning in the context [of] evolutionary [reinforcement learning]" (Koutník et al., 2014). That work used a CNN as part of an evolved controller for an AI player in the TORCS car-racing simulator (Wymann et al., 2014), which took as its input the rendered graphics that a human player would see if they were playing, and reduced that high-dimensionality input down to a feature vector of low dimensionality (Koutník et al., 2014). The CNN was fixed in topology, with only weights evolving. Through use of a novel fitness function, they demonstrated unsupervised learning of the CNN weights, that is, without "using a large training set of class-labeled images via backpropagation" (Koutník et al., 2014). In 2015, Verbancsics and Harguess extended HyperNEAT to evolve weights of an image-classification CNN, still using a fixed CNN topology (Verbancsics & Harguess, 2015).

In 2017, the first works that we know of to use evolutionary methods to generate topologies for CNNs, rather than just weights, were published (Suganuma et al., 2017) (Real et al., 2017) (Xie & Yuille, 2017). These methods could not generate CNNs of arbitrary topology; the first authors to do so seem to be Yuan et al., who introduced the DAGCNN algorithm in 2020, a genetic algorithm with an indirect encoding able to represent different kinds of CNN layers arranged into an arbitrary DAG (Yuan et al., 2020). Their work, and that of the associated teams of authors Irwin-Harris et al. and Sun, Xue, Zhang, and Yen, forms the foundation of this present work, and we will refer to their work frequently in this paper (Irwin-Harris et al., 2019) (Sun, Xue, Zhang, & Yen, 2020b) (Yuan et al., 2020) (Sun, Xue, Zhang, & Yen, 2020a) (Sun, Xue, Zhang, Yen, & Lv, 2020).

2.5. Research Aims

2.5.1 Evaluating the Tendency Hypothesis

Our first goal for this project is to test an assertion that we call the "tendency hypothesis". Recall from Section 2.3 that we said that a generic evolutionary algorithm evaluates the fitness of each individual in each generation. However, in the library that we implement in this project, evaluating the fitness of a single individual involves training a CNN, which is an expensive process.

Instead, when our library wants to choose an individual for reproduction, it uses an algorithm called a "slack binary tournament", shown formally in Algorithm 3.9 on page 35, based on the methods of Sun, Xue, Zhang, and Yen (Sun, Xue, Zhang, & Yen, 2020a). To choose an individual for reproduction by slack binary tournament, we choose, with replacement, two arbitrary individuals from the current generation of individuals. We then train each individual on a given training set. We then use the trained individuals to make predictions on a given validation set, using a given loss function, evaluating the mean and standard deviations of this loss function over the entire set of predictions. The winner of the tournament is the individual with lower average loss, as long as the difference in average loss between the two individuals is greater than some given value, the "mean slack". If the means are within that mean slack value of each other, the winner is the individual with the lower standard deviation of loss, but likewise under the condition that the difference between standard deviations is greater than a given "standard deviation slack". If that condition is not met, the individual with fewer parameters is the winner.

The slack binary tournament reduces the amount of expensive training that our algorithm must do in two ways. The first is that, since we sample with replacement, in general not every individual in a generation need be evaluated. Having calculated

the fitness of an individual once, it is simple to cache it for future reference. The second is that, when training the individuals on the training set, we don't need to train those individuals for as many epochs as we would need to do so if we meant to use them to make useful predictions. As Sun, Xue, Zhang, and Yen put it, it "is sufficient to investigate only the tendency of the performance" (Sun, Xue, Zhang, & Yen, 2020a). In other words, they assert that we can treat the performance of an individual, when trained for a small number of epochs, as an approximation of the performance of that individual when trained for a larger number of epochs, with enough accuracy to be able to use those estimates to distinguish better-performing individuals from worse. The slack values, then, effectively give us a way to specify how much confidence we are willing to place in those estimates. A larger slack value means that, to accept the difference between two estimated values as meaningful, we require the difference between them to be more distinct, allowing more of a "safety margin" between the estimates.

What we investigate in Section 4.1, then, is whether this hypothesis holds, and if so, how many epochs of training should be used for the purposes of conducting slack binary tournaments.

2.5.2 Investigation of Hyperparameters

The behavior of any machine-learning algorithm is governed by a number of parameters; to avoid confusion with the parameters of a given instance of a machine-learning model, we call these parameters of the algorithm itself "hyperparameters" (Li et al., 2021). Given how new this area of research is (as mentioned in Section 2.4), there is little evidence to suggest how to tune the hyperparameters of our algorithm for best results. In Section 4.2, we conduct a series of experiments to learn about the effects of varying the values of these hyperparameters, and combinations

of hyperparameters.

2.5.3 Testing Different Variations on the Base Algorithm

The basic implementation of our algorithm used in the first two series of experiments hews fairly closely to methods introduced by Sun, Xue, Zhang, and Yen; Irwin-Harris et al.; and Yuan et al. In the third series of experiments, Section 4.3, we try a series of variations on the basic algorithm, to see if any significant improvement in the results can be had.

Chapter III. Algorithm Design and Implementation

We developed a library, SDAG, based mainly on the methods of Sun, Xue, Zhang, and Yen; Irwin-Harris et al.; and Yuan et al. (Sun, Xue, Zhang, & Yen, 2020b) (Irwin-Harris et al., 2019) (Yuan et al., 2020) (Sun, Xue, Zhang, & Yen, 2020a) (Sun, Xue, Zhang, Yen, & Lv, 2020), to evolve topologies for CNNs.

Some of the most important classes are briefly described in Table 3.1 on page 24. We will examine each of those classes, and some of their subclasses, in the course of reviewing SDAG's algorithm for evolving CNNs.

3.1. Creating the Initial Generation

The `Population` class is the intended main entry point for programs using this library.

`Population` is generally not meant to be instantiated directly, but rather through the `Population.make_random` factory method (Gamma et al., 1994) (see Algorithm 3.1 on page 24), which initializes a `Population` by repeatedly calling `Genome.make_random` (see Algorithm 3.2 on page 25). `Population.make_random` takes as arguments the shape of a single input example, the number of outputs desired, a training loader, and a validation loader. The loaders are intended to be instances of the `torch.utils.data.DataLoader` class.

As an example, in our experiments evolving CIFAR-10 classifiers below, we

Name	Purpose
`Block`	Subnetwork of an `Individual` corresponding to a single `Gene`.
`Gene`	Specifies the parameters to use for a single `Block` when converting a `Genome` to an `Individual`, which always include the operation performed by the `Block` (e.g. convolution, average pooling, summation), from where the `Block` takes its input, and any parameters specific to the `Block` type (like the kernel size of a convolutional `Block`).
`Genome`	Sequence of `Gene`s that can be converted into an `Individual`. Two "parent" `Genome`s can exchange `Gene`s and produce a "child", in a process called "crossover".
`Individual`	Neural network composed of `Block`s arranged into a directed acyclic graph.
`Mutation`	Changes the `Gene`s in a `Genome`, for example, by removing a `Gene`, inserting a `Gene`, or changing the parameters of a `Gene`.
`Population`	Collection of a single "generation" of `Genome`s.

Table 3.1: Key classes

Algorithm 3.1 `Population.make_random`

1: **function** POPULATION.MAKE_RANDOM(*input_shape*, *n_outputs*, *training_loader*, *validation_loader*, *hyperparameters*)
2: *genomes* ← [ ]
3: **for** *i* ← 1, *hyperparameters.n_genomes* **do**
4: *new_genome* ← GENOME.MAKE_RANDOM(*input_shape*, *n_outputs*, *hyperparameters.min_n_genes*, *hyperparameters.max_n_genes*)
5: *genomes* ← *genomes*.APPEND(*new_genome*)
6: **end for**
7: **return** POPULATION.NEW(*genomes*, *training_loader*, *validation_loader*, *hyperparameters*)
8: **end function**

Algorithm 3.2 `Genome.make_random`

```
1: function GENOME.MAKE_RANDOM(input_shape, n_outputs, min_n_genes,
   max_n_genes)
2:     length ← arbitrary integer in min_n_genes, ..., max_n_genes inclusive
3:     genes ← [ ]
4:     for gene_index ← 0, length − 1 do
5:         gene_class ← arbitrarily chosen concrete subclass of Gene
6:         input_indices ← [ ]
7:         for i ← 1, gene_class.arity do
8:             new_input_index ← arbitrary integer in −1, ..., gene_index − 1
   inclusive
9:             input_indices.APPEND(new_input_index)
10:        end for
11:        if gene_class is ConvGene or DepSepConvGene then
12:            kernel_size ← arbitrary member of gene_class.valid_kernel_sizes
13:            feature_depth ← arbitrary member of
   gene_class.valid_feature_depths
14:            gene ← gene_class.NEW(input_indices, kernel_size, feature_depth)
15:        else
16:            gene ← gene_class.NEW(input_indices)
17:        end if
18:        genes.APPEND(gene)
19:    end for
20:    return Genome.NEW(input_shape, n_outputs, genes)
21: end function
```

want to map 3-channel images that are 32 by 32 pixels in size to one of 10 different classes. Hence we pass `make_random` the input shape (3, 32, 32); 10 for the number of outputs; and two separate `DataLoader`s, one of which we initialize from the 4,500-element training set generated in Section 4.1, and the other from the 500-element validation set.

`Population.make_random` also takes a number of hyperparameters, described in Table 3.2 on page 27. Note that, in our experiments below, we always use the default values for `n_genomes`, `n_generations`, `make_optimizer`, and `criterion_class`, and as such we will not include those hyperparameters when listing the hyperparameters used in an experiment.

3.2. Main Loop

Having created a `Population`, we start the process of evolution by calling `Population.breed`. This method repeatedly, for `n_generations` iterations, produces a new set of `Genome`s from the set of `Genome`s within the `Population`, and replaces the old set with the new set, as in Algorithm 3.3 on page 28.

As we will see in Section 3.2.3 and Section 3.2.4, a fundamental operation in our algorithm for breeding a new generation is the "slack binary tournament". To understand the slack binary tournament, first we must examine SDAG's fitness function.

3.2.1 Evaluating Fitness

As mentioned in Section 2.3.1, we are using an indirect representation for SDAG. This means it is not possible to use a `Genome` directly to make predictions from an input example: a `Genome` is not itself a CNN, but rather a concise description of the topology of one. If we want to actually evaluate data, we must transform a

Name	Description	Default
`n_genomes`	Number of `Genomes` in each generation. Note that it is possible for a `Genome` to appear multiple times in a generation.	100
`n_generations`	Number of generations to run for.	100
`make_optimizer`	Function that takes an `Individual` and returns a `torch.optim.Optimizer` to optimize the `Individual`'s parameters.	`lambda i: Adam(i.parameters())`
`criterion_class`	Loss function class from `torch.nn` module to be used in optimization.	`CrossEntropyLoss`
`min_n_genes`	Minimum number of `Genes` in each randomly-created `Genome` of the initial generation.	10
`max_n_genes`	Maximum number of `Genes` in each randomly-created `Genome` of the initial generation.	15
`elitism_fraction`	When breeding a new generation, fraction of that generation that should be selected by elitism, rather than by crossover.	0.2
`mutation_probability`	When the `Genomes` for a new generation are selected, probability, for each `Gene` in each `Genome`, that that gene will be subjected to a `Mutation`.	0.003
`mean_threshold`	Slack value used when comparing mean losses of two `Genomes` on validation set in slack binary tournament.	0.2
`std_threshold`	Slack value used when comparing standard deviation of losses of two `Genomes` on validation set in slack binary tournament.	0.02

Table 3.2: Available hyperparameters on `Population.make_random`

Algorithm 3.3 `Population.breed`

```
1: procedure POPULATION.BREED
2:     n_by_elitism ← FLOOR(self.n_genomes * self.elitism_fraction)
3:     n_by_crossover ← self.n_genomes − n_by_elitism
4:     for generation_index ← 1, self.n_generations do
5:         new_genomes ← [ ]
6:         for i ← 1, n_by_elitism do
7:             elite_genome ← self.SLACK_BINARY_TOURNAMENT
8:             new_genomes.APPEND(elite_genome)
9:         end for
10:        for i ← 1, n_by_crossover do
11:            parent1 ← self.SLACK_BINARY_TOURNAMENT
12:            parent2 ← self.SLACK_BINARY_TOURNAMENT
13:            child ← parent1.CROSSOVER(parent2)
14:            new_genomes.APPEND(child)
15:        end for
16:        for all genome ∈ new_genomes do
17:            genome.APPLY_MUTATIONS(self.mutation_probability)
18:        end for
19:        self.genomes ← new_genomes
20:    end for
21: end procedure
```

`Genome` into an `Individual` as in Algorithms 3.4 on page 29, 3.5 on page 30, and 3.6 on page 30.

Algorithm 3.4 `Genome.to_individual`

```
1: function GENOME.TO_INDIVIDUAL
2:     blocks ← [ ]
3:     output_shapes ← [ ]
4:     output_indices ← {i : 0 ≤ i < self.genes.length}
5:     for all gene ∈ self.genes do
6:         block ← gene.TO_BLOCK(self.input_shape, output_shapes)
7:         blocks.APPEND(block)
8:         output_shape ← block.OUTPUT_SHAPE
9:         output_shapes.APPEND(output_shape)
10:        for all input_index ∈ gene.input_indices do
11:            output_indices.REMOVE(input_index)
12:        end for
13:    end for
14:    return Individual.NEW(blocks, self.input_shape, output_indices,
       self.output_feature_depth)
15: end function
```

We can thus create an `Individual` based on a given `Genome`. Following the PyTorch convention, the method on the `Individual` that actually takes an example as input and produces a prediction as output is called `forward`. Recall that, in Section 2.1, we described the typical method of training an ANN as descending the gradient of a loss function, using backpropagation to efficiently find the partial derivatives that make up that gradient. A single iteration of the backpropagation algorithm consists of a forward phase, in which input is evaluated by the ANN; and a backward phase, in which the value of the loss function and its partial derivatives are calculated (Aggarwal, 2018). Hence the method name `forward`. Having written `forward`, PyTorch is able to define the corresponding implementation of `backward` automatically, so we don't have to do so explicitly (Paszke et al., 2019). `Block`s also have a `forward` method, which we invoke during `Individual.forward`, but we will not show those in detail here, as they are straightforward implementations of the

Algorithm 3.5 `Gene.to_block`

function GENE.TO_BLOCK(*model_input_shape*, *layer_output_shapes*)
    input_shapes ← [ ]
    for all *input_index* ∈ *self.input_indices* **do**
        if *input_index* = −1 **then**
            input_shapes.append(*model_input_shape*)
        else
            input_shapes.append(*layer_output_shapes*[*input_index*])
        end if
    end for
    block ← *self.block_class*.NEW(*self.input_indices*, *input_shapes*)
    copy *block_class* specific parameters (e.g. convolution layer's filter size) from *self* to *block*
    return *block*
end function

Algorithm 3.6 `Individual.new`

function INDIVIDUAL.NEW(*blocks*, *input_shape*, *output_indices*, *output_feature_depth*)
    self.blocks ← *blocks*
    self.output_indices ← *output_indices*
    self.output_feature_depth ← *output_feature_depth*
    self.tail ← global average-pooling layer followed by fully-connected layer
    initialize fully-connected layer weights with He normal initialization
end function

operations described in Section 2.2.

`Individual.forward` works by iterating over the array of `Block`s within the `Individual`. Call the `Block` b_n which has index n in the `Individual`. We use the input indices of b_n to determine the input values we will use for b_n. If i is an input index of b_n, and $i = -1$, then the corresponding input of b_n uses the *model_input* passed to `Individual.forward`. Otherwise, the input uses the output of b_i. Recall that each input index i of b_n is constrained so that $-1 \leq i < n$, thus ensuring that each input of b_n will be defined by the time we attempt to calculate the output of b_n. If n is in the list of the `Block`'s *output_indices*, we also add the output of b_n to a running list of *tail_inputs*.

Having iterated over all `Block`s, `Individual.forward` then pads the *tail_inputs* as necessary to make them all the same size, and sums them. This sum is then input to the `Individual`'s *tail*, which consists of a global average-pooling layer followed by a fully-connected layer. The global average-pool operation maps a tensor $T \in \mathbb{R}^{c \times x \times y}$ to a tensor $T' \in \mathbb{R}^{c \times 1 \times 1}$ by applying average-pooling with a region size of $x \times y$ to T, thus reducing each plane of T to the mean of all values in that plane. The fully-connected layer maps that $T' \in \mathbb{R}^{c \times 1 \times 1}$ to a vector $\vec{y} \in \mathbb{R}^k$, where k is the number of output classes (hence 10 in the case of CIFAR-10). Each $y_i \in \vec{y}$ is a linear combination of the c scalar values in T', so the fully-connected layer has $c * k$ parameters. $\vec{y}$ is the output from `Individual.forward`.

To evaluate a `Genome`'s fitness, then, as in Algorithm 3.8 on page 33, we first turn it into an `Individual`. In keeping with the results in Section 4.1, we then train the `Individual` for a single epoch on our training set, then use that lightly-trained `Individual` to make predictions for the validation set. We calculate the mean and standard deviation loss (using the loss function specified by the `criterion_class` hyperparameter) for the `Individual` over the validation set, as well as noting down

Algorithm 3.7 `Individual.forward`

```
1: function INDIVIDUAL.FORWARD(model_input)
2:     tail_inputs ← [ ]
3:     block_outputs ← [ ]
4:     for block_index ∈ 0, self.blocks.length − 1 do
5:         block ← self.blocks[block_index]
6:         inputs ← [ ]
7:         for all input_index ∈ block.input_indices do
8:             if input_index = −1 then
9:                 inputs.APPEND(model_input)
10:            else
11:                inputs.APPEND(block_outputs[input_index])
12:            end if
13:        end for
14:        block_output ← block.FORWARD(inputs)
15:        block_outputs.APPEND(block_output)
16:        if block_index ∈ self.output_indices then
17:            tail_inputs.APPEND(block_output)
18:        end if
19:    end for
20:    tail_inputs ←MATCH_SHAPES(tail_inputs)
21:    tail_input ← sum of all tail_inputs
22:    return self.tail.FORWARD(tail_input)
23: end function
```

how many parameters it has. These values are cached for each `Genome` by the `Population`, so that if the same `Genome` appears again in the current generation or a future one, we don't have to repeat the expensive computations involved in evaluating that `Genome`. As we will see in Section 3.2.3, due to elitism, we can expect a significant number of cache hits.

Algorithm 3.8 `Population.evaluate_fitness`

function POPULATION.EVALUATE_FITNESS(*genome*)
    individual ← *genome*.TO_INDIVIDUAL
    $loss\_function \leftarrow self.criterion\_class$.NEW
    optimizer ← $self$.MAKE_OPTIMIZER(individual)
    $training\_data \leftarrow self.training\_loader.data$
    $training\_labels \leftarrow self.training\_loader.labels$
    $validation\_data \leftarrow self.validation\_loader.data$
    $validation\_labels \leftarrow self.validation\_loader.labels$
    individual.TRAIN_ONE_EPOCH(*training_data*, *training_labels*, $loss\_function$, *optimizer*)
    $validation\_predictions \leftarrow individual$.FORWARD(validation_data)
    losses ← LOSS_FUNCTION(*validation_predictions*, *validation_labels*)
    return {"mean": MEAN(*losses*), "std": STANDARD_DEVIATION(*losses*), "n_parameters": $individual.n\_parameters$}
end function

3.2.2 Slack Binary Tournament

As stated above in Section 3.2, a key operation when breeding a new generation is the slack binary tournament, shown in Algorithm 3.9 on page 35 (Sun, Xue, Zhang, & Yen, 2020a). It is used by other parts of the algorithm–elitism and crossover, discussed below in Sections 3.2.3 and 3.2.4–to choose `Genome`s on which to operate.

To choose a `Genome` by slack binary tournament, we first choose with replacement two arbitrary `Genome`s, with uniform probability, from the `Population`. We get the fitness of each, either computed via Algorithm 3.8 on page 33 or looked up in the cache. Recall that the fitness value contains three seperate metrics: the mean

loss, the standard deviation of loss, and the number of parameters. Between both `Individuals`, we compare the difference in the mean loss. If that difference is greater than the `mean_threshold` parameter, the `Genome` corresponding to the `Individual` with the lower mean loss is the winner of the tournament. Otherwise, we compare the difference in the standard deviation loss, and if that difference is greater than the `std_threshold` hyperparameter, the winner is the `Genome` corresponding to the `Individual` with the lower standard deviation loss. If neither of those two conditions apply, the `Genome` whose `Individual` has fewer parameters wins.

One notable difference between our implementation of slack binary tournaments and that of Sun, Xue, Zhang, and Yen (Sun, Xue, Zhang, & Yen, 2020a) is in how we break ties when the difference in mean loss between the `Genomes` is smaller than the mean threshold, i.e. when the conditionals on lines 16 and 18 of Algorithm 3.9 on page 35 both evaluate to false. In their implementation, when this is the case, they proceed to compare first the number of parameters, and then the standard deviations of loss (with slack). We have reversed this order, comparing first standard deviation, and then only if necessary the number of parameters. Sun, Xue, Zhang, and Yen's reasoning for making the parameter count more important than the standard deviation is that networks with fewer parameters consume less of the limited power and compute budgets of mobile devices. Smartphones with built-in high-resolution digital cameras and a reasonable amount of computing power are now ubiquitous, suggesting many possibilities for CNN-based mobile apps, so those authors do make a good point in identifying the feasability of implementing a given network on a mobile device as a potentially important consideration. Moreover, as engineers working in an age of anthropogenic climate crisis, in a field that consumes a significant portion of the world's generated electricity, it is morally incumbent upon us to consider the energy efficiency of our software.

Algorithm 3.9 `Population.slack_binary_tournament`

```
1: function POPULATION.SLACK_BINARY_TOURNAMENT
2:     genome1 ← self.PICK_ARBITRARY_GENOME( )
3:     genome2 ← self.PICK_ARBITRARY_GENOME( )
4:     if genome1 ∈ self.fitness_cache then
5:         genome1_fitness ← self.fitness_cache[genome1]
6:     else
7:         genome1_fitness ← self.EVALUATE_FITNESS(genome1)
8:         self.fitness_cache[genome1] ← genome1_fitness
9:     end if
10:    if genome2 ∈ self.fitness_cache then
11:        genome2_fitness ← self.fitness_cache[genome2]
12:    else
13:        genome2_fitness ← self.evaluate_fitness(genome2)
14:        self.fitness_cache[genome2] ← genome2_fitness
15:    end if
16:    if genome1_fitness.mean − genome2_fitness.mean ≥
   self.hyperparameters.mean_threshold then
17:        return genome2
18:    else if genome2_fitness.mean − genome1_fitness.mean ≥
   self.hyperparameters.mean_threshold then
19:        return genome1
20:    else if genome1_fitness.std − genome2_fitness.std ≥
   self.hyperparameters.std_threshold then
21:        return genome2
22:    else if genome2_fitness.std − genome1_fitness.std ≥
   self.hyperparameters.std_threshold then
23:        return genome1
24:    else if genome1_fitness.n_parameters < genome2_fitness.n_parameters
   then
25:        return genome1
26:    else
27:        return genome2
28:    end if
29: end function
```

However, in this work, we are primarily interested in validating (or not) the effectiveness of the solutions produced by this approach to topology search. There is not much point in optimizing an algorithm that doesn't produce usable results in the first place. Unlike the number of parameters, and like the mean loss, the standard deviation of loss for a given network is a measurement of the quality of predictions made by that network. Informally we can think of the standard deviation as measuring the consistency of this quality, and for most applications we would prefer a network that can consistently make good predictions to one that often makes both extremely good and extremely bad predictions, even though they might have similar mean losses. Hence our decision to give the standard deviation priority over parameter count.

3.2.3 Elitism

Some subset of the replacement set of `Genome`s in each iteration of `Population.breed` is chosen through a process called "elitism". The proportion of replacements so chosen is controlled by the `elitism_fraction` hyperparameter. To use the evolutionary terms from Section 2.3, elitism is like asexual reproduction, in which a single individual's genome is copied.

Elitism simply consists of conducting multiple slack binary tournaments on the current generation of `Genome`s, and keeping the winners of each tournament unchanged for the next generation. For example, if we use the default values of 0.2 for `elitism_fraction` and 100 for `n_genomes`, when breeding a new generation, the `Population` will conduct 20 slack binary tournaments, keeping the winners of those. Note that, since we sample with replacement in a slack binary tournament, it is possible that a given `Genome` will compete in and possibly win multiple tournaments during elitism, so generally a `Genome` can appear multiple times in a `Population`, and

the number of distinct `Genome`s may be less than `n_genomes`. Also, although we copy the winning `Genome`s unchanged into the new generation during this step, they may still be altered through mutation, as described below in Section 3.2.5.

3.2.4 Crossover

Crossover is SDAG's analogue of sexual reproduction, in which the genomes of two individuals mingle together to produce a child with similarities to both parents. As shown in Algorithm 3.10 on page 38, in a single crossover operation, we first conduct two separate slack binary tournaments to select two "parent" `Genome`s, A and B. Define the function $l(G)$ as returning the length in `Gene`s of the `Genome` G, and define $G[n]$, with $0 \leq n < l(G)$, as returning the n^{th} `Gene` in G. We will assume, without loss of generality, that $l(A) \leq l(B)$. We then choose random integers i and j such that $0 \leq i < j \leq l(A)$. With equal probability, we designate one of A and B to be the "outside" parent O, and the other to be the "inside" parent I. We then construct a "child" genome C such that $l(C) = l(O)$, and:

$$C[n] = \begin{cases} I[n] & \text{if } i \leq n < j \\ O[n] & \text{otherwise} \end{cases} \tag{3.1}$$

This C is then the result of the crossover operation.

If we were again to use the default values of 0.2 for `elitism_fraction` and 100 for `n_genomes`, when breeding a new generation, we will then generate $(1-0.2)*100 = 80$ such children, conducting 160 slack binary tournaments in the process.

3.2.5 Mutation

We now have `n_genomes Genome`s prepared through a mixture of elitism and crossover. The final step in creating the new generation is to apply `Mutation`s to

Algorithm 3.10 `Genome.crossover`

```
 1: function GENOME.CROSSOVER(other)
 2:     n_genes ← self.genes.length
 3:     if n_genes > other.genes.length then
 4:         return other.CROSSOVER(self)
 5:     end if
 6:     start_index ← arbitrary integer from 0 to n_genes − 1 inclusive
 7:     end_index ← arbitrary integer from start_index + 1 to n_genes inclusive
 8:     coin_flip ← either 0 or 1 with equal probability
 9:     if coin_flip = 0 then
10:         outer_parent = self
11:         inner_parent = other
12:     else
13:         outer_parent = other
14:         inner_parent = self
15:     end if
16:     head ← outer_parent.genes[0 : start_index]
17:     middle ← inner_parent.genes[start_index : end_index]
18:     tail ← outer_parent.genes[end_index : outer_parent.genes.length]
19:     child_genes ← CONCATENATE_LISTS(head, middle, tail)
20:     child = Genome.NEW(self.input_shape, self.output_feature_depth, child_genes)
21:     return child
22: end function
```

Name	Allowed on	Effect
`ChooseInputMutation`	All `Gene`s	Replace one of the target `Gene`'s input indices with an arbitrarily chosen valid index.
`DeletionMutation`	All `Gene`s	Remove the target `Gene` from the `Genome`, unless that would leave the `Genome` empty.
`InsertionMutation`	All `Gene`s	Insert a randomly-created `Gene` before or after the target `Gene`.
`ChooseKernelSizeMutation`	`ConvGene`, `DepSepConvGene`	Replace the target `Gene`'s kernel size with an arbitrarily chosen valid value.
`ChooseOutputFeatureDepthMutation`	`ConvGene`, `DepSepConvGene`	Replace the target `Gene`'s output feature depth with an arbitrarily chosen valid value.

Table 3.3: Available `Mutation`s

these `Genome`s. We apply an arbitrarily-selected `Mutation` to every `Gene` in every `Genome` with probability `mutation_probability`. The available `Mutation`s are listed in Table 3.3 on page 39.

Each of the concrete subclasses of `Mutation` implement an `apply` method, which takes a single `Gene` as input and returns a list (possibly empty) of `Gene`s to replace the input gene with in the `Genome`. Hence, `DeletionMutation.apply` always returns an empty list, `InsertionMutation` returns a list consiting of the original `Gene` with a new gene before or after it, and the other `Mutation`s return a list whose single element is a modified version of the input `Gene`. This process is shown in Algorithm 3.11 on page 40. In the case that no mutation is applied, we return a list containing only the original `Gene`. Doing this lets us simplify other parts of Algorithm 3.11.

One point to note is that, if a `Mutation` adds or deletes `Gene`s, it's possible that input indices of subsequent `Gene`s may point to different sources than before

Algorithm 3.11 `Genome.apply_mutations`

```
1: function GENOME.APPLY_MUTATIONS(mutation_probability)
2:     new_genes ← [ ]
3:     adjustments ← array with value 0 repeated self.genes.length times
4:     for source_index ← 0, self.genes.length − 1 do
5:         source_gene ← self.genes[source_index]
6:         if arbitrary real number between 0 and 1 ≤ mutation_probability then
7:             mutation ← source_gene.VALID_MUTATION(source_index)
8:             replacement_genes ← mutation.APPLY(source_gene)
9:         else
10:            replacement_genes ← [source_gene]
11:        end if
12:        for all replacement_gene ∈ replacement_genes do
13:            new_input_indices ← [ ]
14:            for all input_index ∈ replacement_gene.input_indices do
15:                if input_index = −1 then
16:                    new_input_indices.APPEND(−1)
17:                else
18:                    adjustment ← adjustments[input_index]
19:                    new_input_indices.APPEND(input_index + adjustment)
20:                end if
21:            end for
22:            replacement_gene.input_indices ← new_input_indices
23:        end for
24:        adjustment_change ← replacement_genes.length − 1
25:        for adjustment_index ← source_index, self.genes.length − 1 do
26:            adjustments[adjustment_index] ← adjustments[adjustment_index] +
    adjustment_change
27:        end for
28:        new_genes ← CONCATENATE_LISTS(new_genes, replacement_genes)
29:    end for
30:    if new_genes ==[ ] then
31:        new_genes ← self.genes
32:    end if
33:    return Genome.NEW(self.input_shape, self.output_feature_depth, new_genes)
34: end function
```

the `Mutation` was applied. For example, if we delete the `Gene` at index k, then each `Gene`s with original index $i, i > k$ will wind up with index $i - 1$; inserting a `Gene` has the opposite effect. We address this by maintaining an array of these `adjustments` of the same length as the `Genome`, which starts out initialized to all 0s (i.e. no adjustment needed). For each source gene with original index k, after we've gotten the corresponding list of `replacement_genes`, we update the `input_indices` of each replacement gene by adding to each the appropriate adjustment from `adjustments`. The pertinent adjustments are then updated based on the length of `replacement_genes`.

There is one significant difference between our implementation of these input index adjustments and that of Yuan et al. which originally noted this question of changing inputs (Yuan et al., 2020). In Yuan et al.'s algorithm, when a mutation inserts genes, affected input indices in genes that follow are only updated with 50% probability, whereas we always make the relevant adjustments. Their algorithm does, like ours, always apply adjustments in the case that a gene is deleted.

3.2.6 End of the Main Loop

With the mutations applied, the `Genome`s for the next generation are ready, and we replace the contents of the `Population` with these new `Genome`s. This completes one iteration of `Population.breed`. If all goes well, after running for `n_generations` iterations, we will end up with `Genome`s representing much fitter `Individual`s than we began with.

Chapter IV. Experiments and Results

4.1. Evaluating the Tendency Hypothesis

A series of experiments was performed to test the "tendency hypothesis", as described in Section 2.5. A secondary goal of these experiments was to investigate the relationship, if any, between genome length and the optimal number of epochs of training.

All experiments in this section were conducted on a laptop PC with four 3.9 GHz Intel i7-7820HK CPU cores, 32 GiB of RAM, and an nVidia GeForce GTX 1070 Mobile GPU with 8 GiB of memory. The operating system was Ubuntu Linux 20.04. The experiments were implemented in Python 3.9.5, using the PyTorch machine learning library version 1.8.1+cu102 (Paszke et al., 2019) and CUDA 11.2 (Nickolls et al., 2008).

The CIFAR-10 dataset (Krizhevsky, 2009b) was used for these experiments. CIFAR-10 consists of a training dataset of 50,000 images, and a test dataset of 10,000 images. All images are color bitmaps with each pixel represented by a triplet of bytes giving RGB values between 0 and 255 inclusive. Images are 32 pixels by 32 pixels in size. Images are labelled with one of 10 class labels. The classes are balanced perfectly: that is, the training dataset contains 5,000 members of each different class, and the test dataset 1,000.

For efficiency, following the practice of Irwin-Harris et al. (Irwin-Harris et al.,

2019), we do not use the full CIFAR-10 dataset for these tests. Rather, from the CIFAR-10 training dataset, 450 elements of each of the 10 classes were selected to form a 4,500 element training set. A validation set of 500 elements, disjoint from the training set and likewise balanced between classes, was also selected. The training and validation sets were both normalized by subtracting the mean and dividing by the standard deviation of the full 50,000 element training set.

In each experiment, a population of 100 random CNNs was generated by the method in Algorithm 3.1 on page 24. The genome length varied by experiment, ranging from 5 to 30 in increments of 5–hence six experiments in total. Each CNN was trained for 100 epochs on the training set. Minibatches consisted of 50 elements, which number was chosen based on the memory limitations of the available hardware. The cross-entropy loss function (Aggarwal, 2018) (Bridle, 1990) was used, and minimized by an Adam optimizer (Kingma & Ba, 2017), with parameters $\alpha = 0.01, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1\text{e} - 8$.

For a given experiment, let $\{i_0, ..., i_{99}\}$ be the population of CNNs. Define $t_n = \{t_{n,0}, ..., t_{n,100}\}$, with each $t_{n,m}$ being the average loss over the training set for i_m after n epochs of training. Define v_n similarly for the validation set.

In Figures 4.1 on page 44 and 4.2 on page 45, we plot the average of each t_n and v_n, for n between 1 and 100 inclusive. We can see that, in each experiment, the average loss on the training set continues to descend with every epoch to or almost to the end of the experiment. However, in each case, the average loss on the validation set descends for a number of epochs, then rises again, suggesting that past that number of epochs the models are overfitting on the training set.

For each experiment, let m be the number of epochs that minimizes the average loss on the validation set v_n. The values of m and corresponding minimum of v_n is given in Table 4.1 on page 46. For each $n \in [1, ..., m-1]$, we calculated the correlations

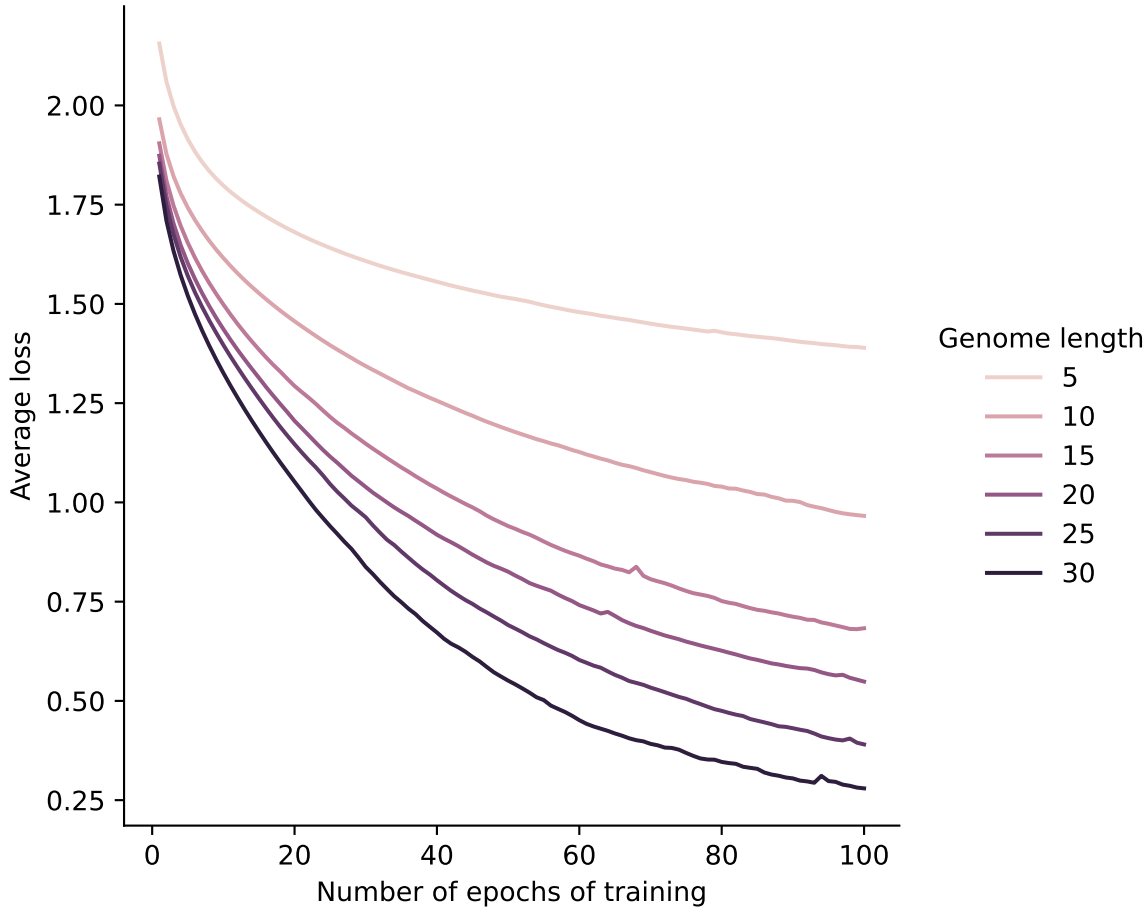

Figure 4.1: Average loss on training set

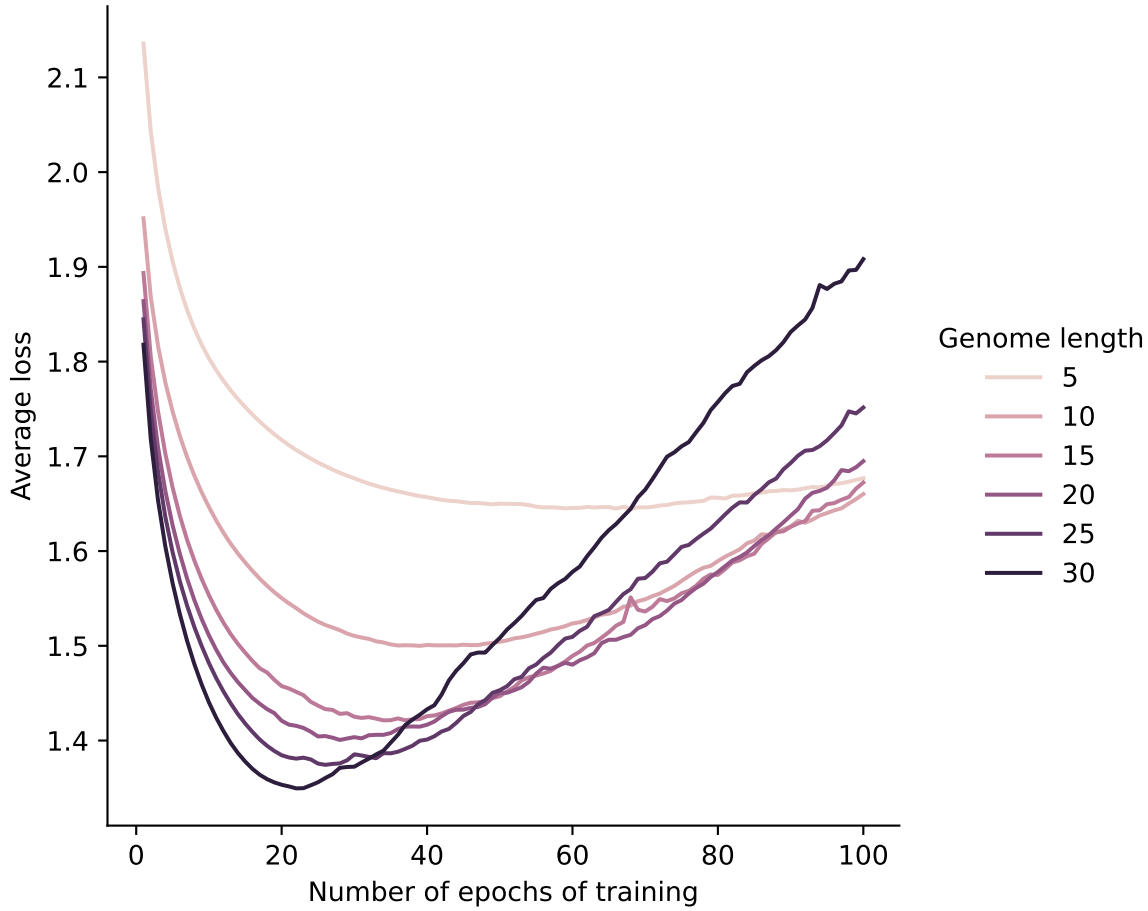

Figure 4.2: Average loss on validation set

Genome length	m	v_m
5	68	1.645
10	39	1.500
15	34	1.421
20	28	1.401
25	26	1.374
30	22	1.350

Table 4.1: Minimum average validation set losses by genome length

$\rho(t_n, v_n)$; $\rho(t_n, v_m)$; and $\rho(v_n, v_m)$. These correlations are plotted in Figures 4.3–4.5 on pages 47, 48, and 49 respectively.

Past the first few epochs, we see sharp declines in $\rho(t_n, v_n)$ and even more so in $\rho(t_n, v_m)$, which suggests that loss on the training set is only a weak predictor of loss on the validation set. In contrast, $\rho(v_n, v_m)$, in most cases, exhibits a pattern of a sharp increase in the first few epochs, followed by a plateau, followed by another sharp increase as n goes to m and $\rho(v_n, v_m)$ goes to 1. In Figure 4.6 on page 50, we consider only $n \in [1, 10]$, and "zoomed in" the increase in $\rho(v_n, v_m)$ in early epochs looks less precipitous.

Based on these findings, if we have an estimate of m for a given model, and we would like to estimate v_m for that model by training the model on the training set for a given number of epochs and then measuring the validation set loss, two strategies suggest themselves. One is to train the model for m epochs or close to it, and accept that the higher accuracy in our estimation of v_m comes at a performance cost. The other strategy is to train the model for very few epochs, perhaps just one, and accept that the high performance of this strategy comes at the cost of accuracy in our estimation of v_m.

The values of $\rho(v_1, v_m)$ are shown in Table 4.2 on page 47, from which we

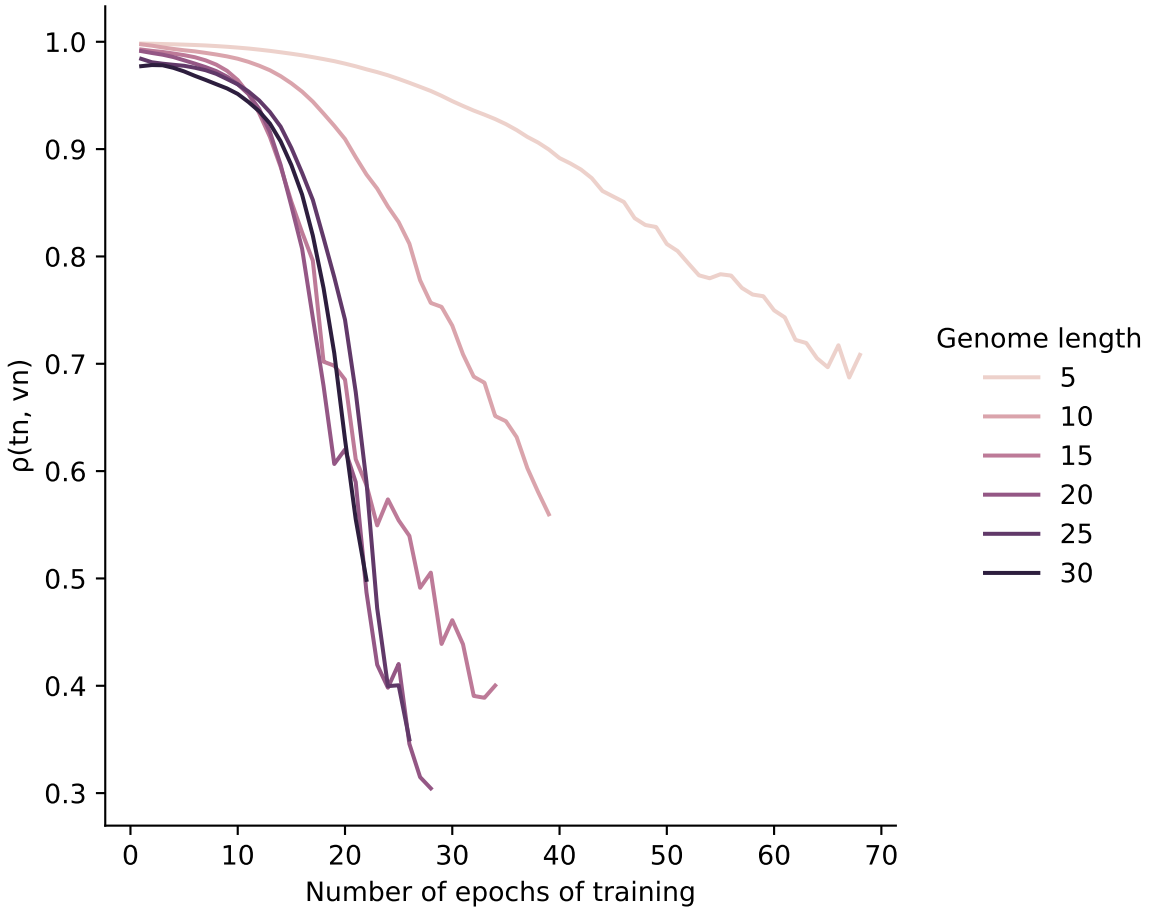

Figure 4.3: $\rho(t_n, v_n)$

Genome length	m	$\rho(v_1, v_m)$
5	68	0.615
10	39	0.573
15	34	0.539
20	28	0.540
25	26	0.550
30	22	0.625

Table 4.2: $\rho(v_1, v_m)$ by genome length

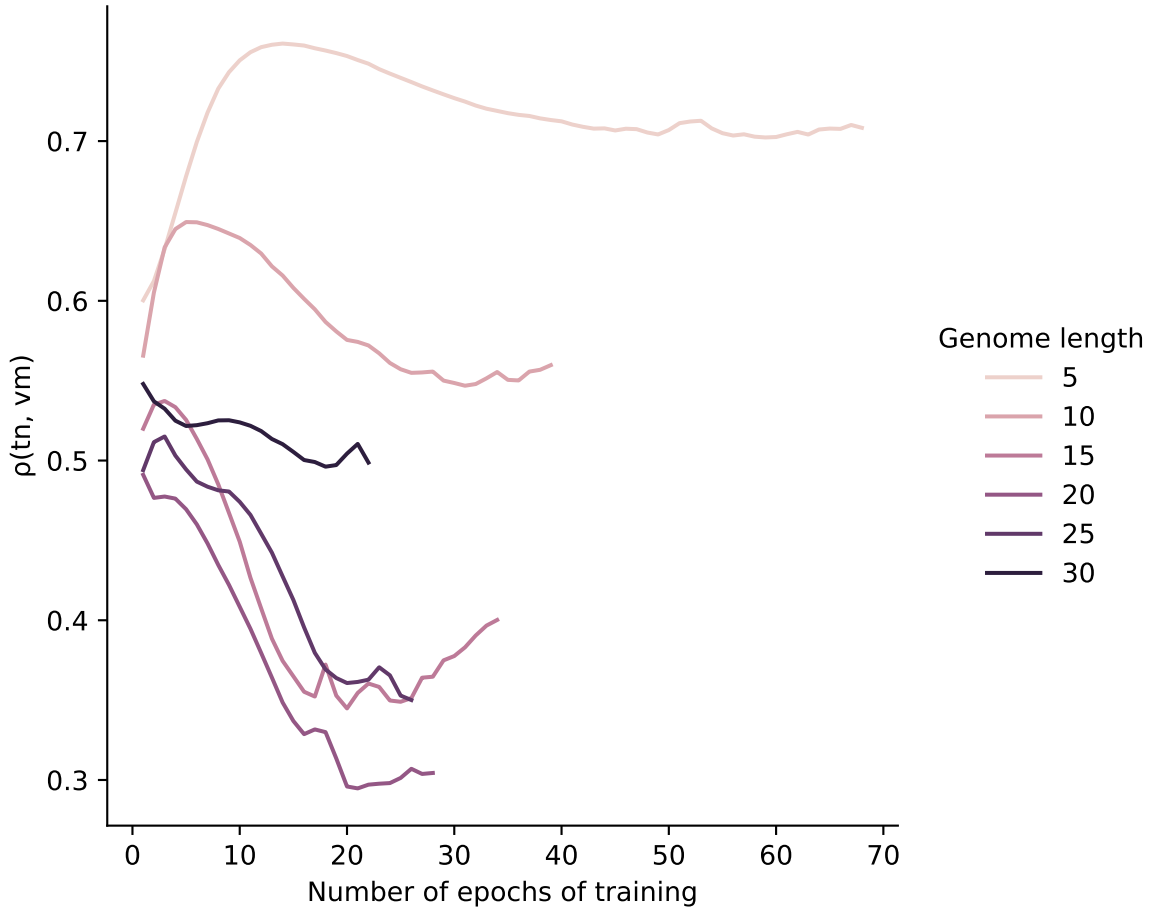

Figure 4.4: $\rho(t_n, v_m)$

Genome length	m	$\rho(t_1, v_m)$
5	68	0.600
10	39	0.566
15	34	0.520
20	28	0.491
25	26	0.494
30	22	0.548

Table 4.3: $\rho(t_1, v_m)$ by genome length

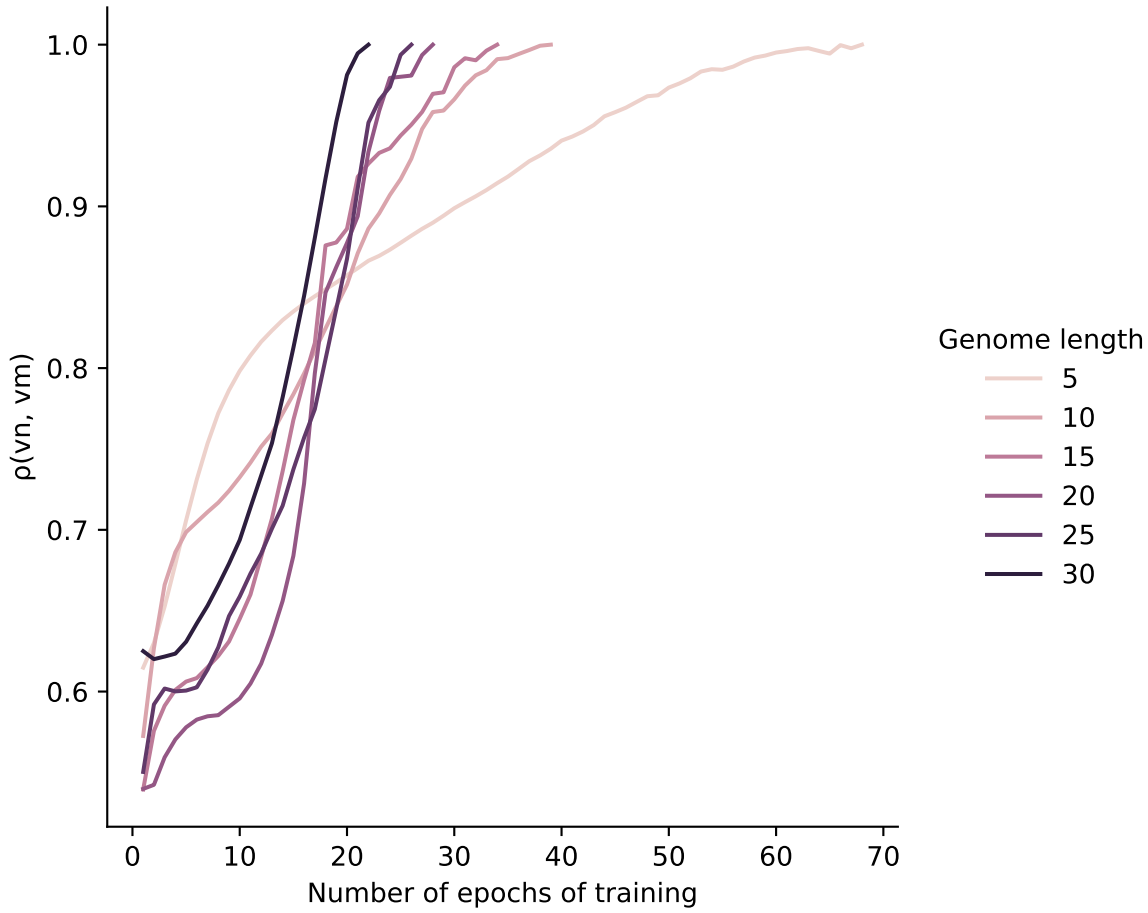

Figure 4.5: $\rho(v_n, v_m)$

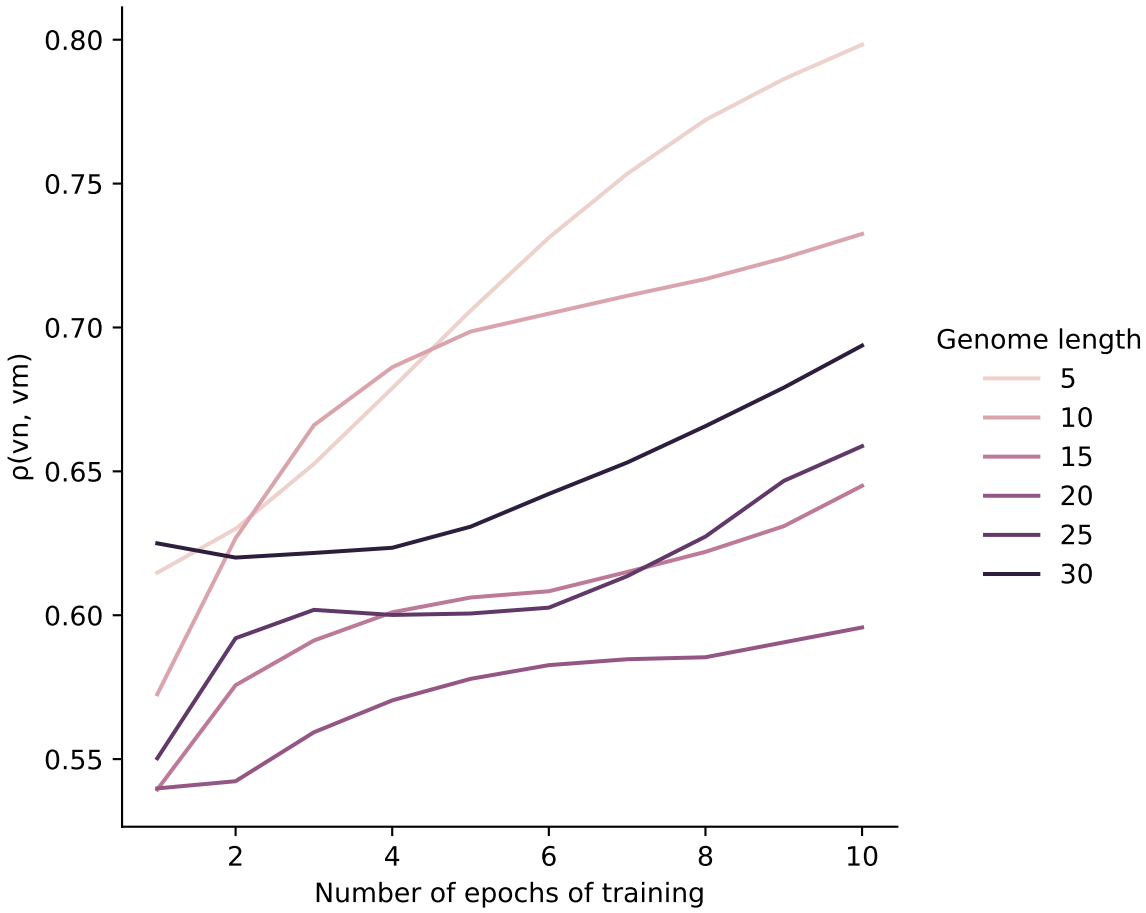

Figure 4.6: $\rho(v_n, v_m)$ for $1 \leq n \leq 10$

can see that the measured correlations all fell in the range 0.582 ± 0.043. In our evolutionary experiments below, when we compare the estimated validation set loss between two models, we do so using slack binary tournament selection (see Section 3.2.2), which is designed to tolerate inaccuracy in the estimated losses of the models. Given these, it seems reasonable to use the single-training-epoch strategy when estimating a model's validation loss. We will expect that, in comparison with the many-training-epochs strategy, our evolved models will take more generations of evolution to reach an acceptable level of performance, but each generation will require less wall-clock time. Another advantage of this method is that it relieves us of having to know (or estimate) m ahead of time.

Could we estimate the minimal average validation loss from average training loss after one epoch of training, ignoring the validation set entirely? Comparing Table 4.2 on 47 to Table 4.3 on page 48, we see that, while $\rho(t_1, v_m)$ is well above zero in every case, it is also less than $\rho(v_1, v_m)$ in every case. Generally, a training set for a given problem will be larger than a validation set, so we would expect calculating the average loss over the training set to take more computation than for the validation set. Given that it would also lead to worse estimates of v_m, we can safely ignore this approach.

One avenue of future investigation would be to repeat these experiments, but rather than using randomly-generated CNNs, using CNNs refined through the evolutionary process in Chapter 3, to see if that leads to different results.

Hyperparameter(s)	Possible values
(Minimum number of genes, maximum number of genes)	(10, 20), (20, 30)
Elitism fraction	0.2, 0.6
Mutation probability	0.1, 0.001
(Mean threshold, standard deviation threshold)	(0.1, 0.01), (0.4, 0.04), (1.6, 0.16)

Table 4.4: Hyperparameters varied by grid search in CIFAR-10 experiment 1

4.2. Investigation of Hyperparameters

4.2.1 Experiment 1

Our first experiment in evolving CIFAR-10 classifiers used the same hardware and software stacks as in Section 4.1. Given both the modest power of that machine, and our lack of experience in tuning the hyperparameters for this method, this experiment consisted of a grid search (Hutter et al., 2019) with few possible values for each hyperparameter, but large differences between the values. In this way we hoped to roughly determine what regions of the hyperparameter space produce good results, with the intent of doing finer-grained explorations in those regions.

The batch size, loss function, and optimizer were all as in Section 4.1. We also re-used the same 4,500-element training set and 500-element validation set from that experiment. Each `Population` was evolved for 100 generations.

Table 4.4 on page 52 shows the different hyperparameter values used by the grid search. Note that, in the case of minimum and maximum genome length, and in that of the mean and standard deviation thresholds, both values are varied in concert. Given that we have for these four sets of hyperparameters respectively 2 sets of values, 2 sets of values, 2 sets of values, and 3 sets of values, this yields 2 * 2 * 2 * 3 = 24 unique combinations of hyperparameters, and thus this experiment evaluated 24 separate `Populations`.

In Table 4.5 on page 56, we have listed the hyperparameters of each `Population` used in experiment 1. We assigned a maximum accuracy score between 0 and 1 inclusive to each `Population`, and have sorted the table in order of descending maximum accuracy. To compute the maximum accuracy for a `Population`, we took the `Individuals` making up the `Population`, and trained each for 50 epochs on the same 4,500-element training set used for evolving the `Populations`. After every 5^{th} epoch, we calculated the accuracy of the `Individual` on classifying the full 10,000-element test set. We did this, rather than simply running for a certain number of epochs and calculating accuracy once, to inform our eventual estimation of how many epochs of training is best. Thus, if a `Population` contained n `Individuals`, it was associated with $10n$ accuracy scores, of which we use the highest as the `Population`'s maximum accuracy score. The topology of the most accurate `Individual` is shown in Figure 4.7 on page 54.

Note that, while accuracy is a simple and intuitive metric, it can also be misleading in the common case that the classes in the data are imbalanced (Juba & Le, 2019). Imagine testing a classifier on a dataset of which 99% of examples belong to Class A, and 1% belong to Class B. A classifier that ignored its input and predicted Class A for all examples would achieve 99% accuracy, and yet have no value for making predictions. In this case, CIFAR-10 is balanced perfectly between classes, as are the working subsets we selected from it, so it is safe to use accuracy.

Examining Table 4.5 on page 56, one thing becomes quickly clear. `Populations` with mean threshold of 0.1 and standard deviation threshold of 0.01 all scored better than `Populations` with higher threshold values. Furthermore, if we calculate the difference in accuracy between each `Population` and the next-worst on the table, we see that by far the single biggest decline in accuracy is between the worst of the low-threshold-value `Populations`, and the best of the high-threshold-value `Populations`.

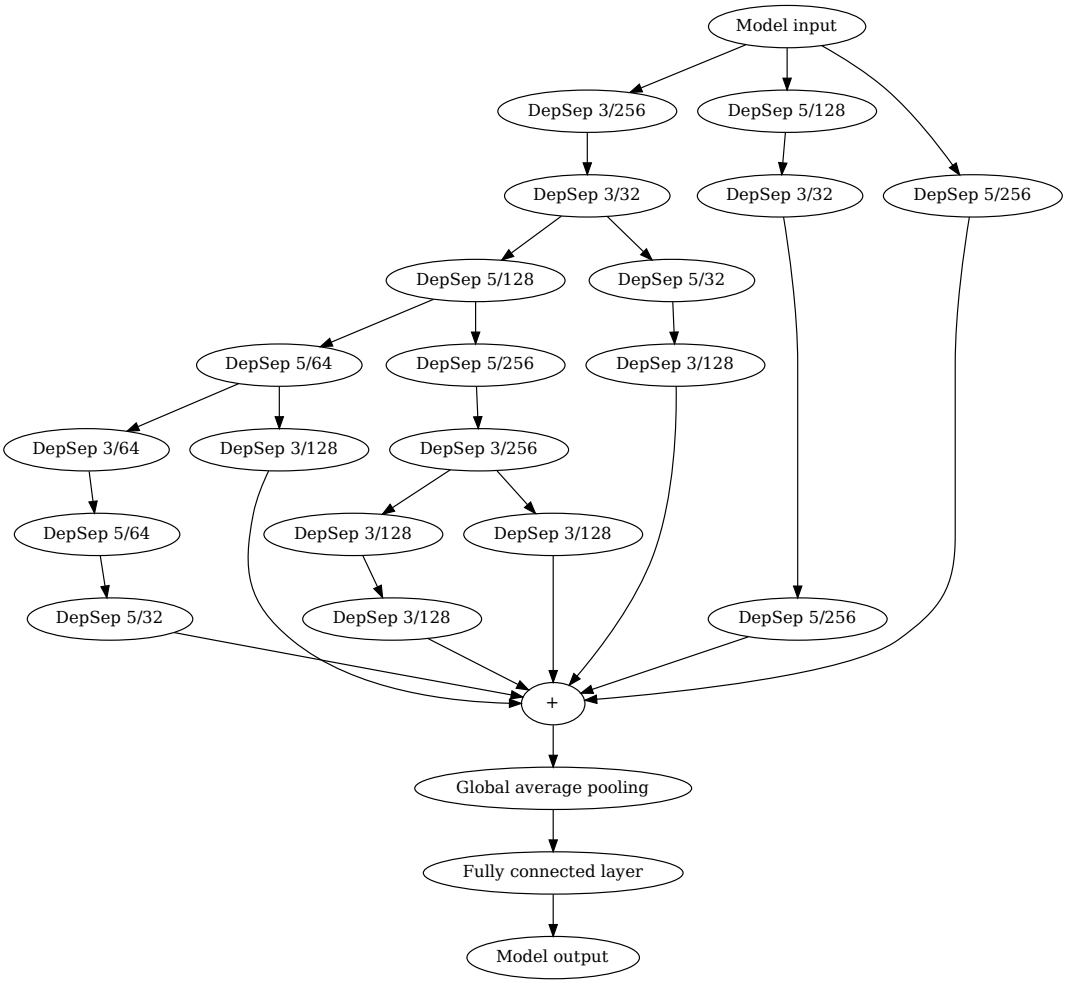

Figure 4.7: Topology of best-performing network from experiment 1. "Conv n/d" indicates a convolutional block with an n by n kernel and d output feature maps. "DepSep n/d" indicates likewise for depthwise separable convolutional blocks.

It would seem that the threshold values have significant influence on the final accuracy score, and furthermore that the optimal threshold values might be lower than any we tried in experiment 1. This suggests a clear direction for experiment 2.

One thing in these results that ran contrary to our intuition is that varying the range of possible lengths of `Genome`s in the initial `Population` via `min_n_genes` and `max_n_genes` between [10...20] and [20...30] had little effect on maximum accuracy. We would have thought that, all else being equal, longer `Genome`s would lead to higher accuracy, but this does not seem to have happened. Since there seems to be no advantage to using the larger range, and longer `Genome`s require more computation than comparable shorter ones to evaluate, we will only use that lower range in further experiments for the time being.

4.2.2 Experiment 2

From this experiment on, we had a much more powerful computer available than in earlier experiments. This new computer was a desktop PC with six 4.9 GHz Intel i5-11600K CPU cores, 32 GiB of RAM, and an nVidia GeForce RTX 3090 GPU with 24 GiB of memory. The operating system was again Ubuntu Linux 20.04. Python 3.9.5 was once again used, with version 1.9.0+cu111 of PyTorch (Paszke et al., 2019) and CUDA 11.4 (Nickolls et al., 2008).

Based on the results of experiment 1, our goal for this experiment was to find an appropriate order of magnitude for the mean threshold and standard deviation threshold hyperparameters. Another grid search was carried out, using the same method as in Section 4.2.1. However, as shown in Table 4.6 on page 57, in this experiment we held all hyperparameters constant except for the thresholds. We used the mean and standard deviation thresholds of 0.1 and 0.01 respectively from the earlier experiment as our starting point, and descend logarithmically from there down

Maximum accuracy	Minimum # genes	Maximum # genes	Elitism fraction	Mutation probability	Mean threshold	Standard deviation threshold
0.622	10	20	0.2	0.1	0.1	0.01
0.610	20	30	0.2	0.1	0.1	0.01
0.607	20	30	0.6	0.1	0.1	0.01
0.606	10	20	0.2	0.001	0.1	0.01
0.605	20	30	0.6	0.001	0.1	0.01
0.603	10	20	0.6	0.1	0.1	0.01
0.600	20	30	0.2	0.001	0.1	0.01
0.590	10	20	0.6	0.001	0.1	0.01
0.402	20	30	0.6	0.1	0.4	0.04
0.384	20	30	0.2	0.1	0.4	0.04
0.369	10	20	0.2	0.1	0.4	0.04
0.358	10	20	0.6	0.1	0.4	0.04
0.265	10	20	0.6	0.001	0.4	0.04
0.222	20	30	0.2	0.001	0.4	0.04
0.221	20	30	0.2	0.001	1.6	0.16
0.218	10	20	0.2	0.001	1.6	0.16
0.218	20	30	0.6	0.001	1.6	0.16
0.216	20	30	0.6	0.001	0.4	0.04
0.204	10	20	0.6	0.1	1.6	0.16
0.198	10	20	0.2	0.1	1.6	0.16
0.185	20	30	0.2	0.1	1.6	0.16
0.184	10	20	0.6	0.001	1.6	0.16
0.182	20	30	0.6	0.1	1.6	0.16
0.166	10	20	0.2	0.001	0.4	0.04

Table 4.5: Performance of evolved CIFAR-10 classifiers, experiment 1

Hyperparameter(s)	Possible values
(Minimum number of genes, maximum number of genes)	(10, 20)
Elitism fraction	0.2
Mutation probability	0.1
(Mean threshold, standard deviation threshold)	(0.1, 0.01), (0.01, 0.001), (0.001, 0.0001), (0.0001, 0.00001)

Table 4.6: Hyperparameters varied by grid search in CIFAR-10 experiment 2

Maximum accuracy	Minimum # genes	Maximum # genes	Elitism fraction	Mutation probability	Mean threshold	Standard deviation threshold
0.685	10	20	0.2	0.1	0.01	0.001
0.673	10	20	0.2	0.1	0.0001	0.00001
0.672	10	20	0.2	0.1	0.001	0.0001
0.622	10	20	0.2	0.1	0.1	0.01

Table 4.7: Performance of evolved CIFAR-10 classifiers, experiment 2

to 0.0001 and 0.00001 respectively. For the other hyperparameters, we used the same values as in the `Population` from experiment 1 with the highest maximum accuracy score. Thus, this experiment only evolved four separate `Population`s.

The results are summed up in Table 4.7 on page 57, with Figure 4.8 on page 58 showing the best generated topology. Comparing this to Table 4.5 on page 56, we see that, consistent with our speculations after experiment 1, the hyperparameters that yielded the best results in experiment 1 yield significantly inferior results to values an order or orders of magnitude lower. Note also that the accuracy calculated for that set of hyperparameters is equal to three significant figures in both experiments. The actual difference is roughly $2 * 10^{-4}$, i.e. 0.002%.

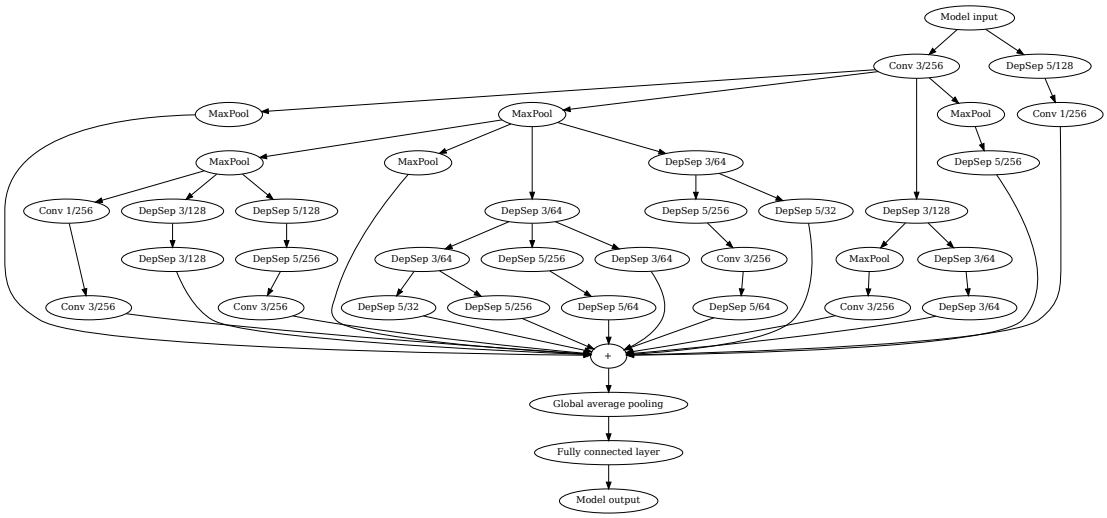

Figure 4.8: Topology of best-performing network from experiment 2. "Conv n/d" indicates a convolutional block with an n by n kernel and d output feature maps. "DepSep n/d" indicates likewise for depthwise separable convolutional blocks.

Hyperparameter(s)	Possible values
(Minimum number of genes, maximum number of genes)	(10, 20)
Elitism fraction	0.05, 0.1, 0.2, 0.4, 0.8
Mutation probability	0.1, 0.01, 0.001
(Mean threshold, standard deviation threshold)	(0.01, 0.001)

Table 4.8: Hyperparameters varied by grid search in CIFAR-10 experiment 3

4.2.3 Experiment 3

Experiment 3 was an investigation of the effect of varying only `elitism_fraction` and `mutation_probability`. Otherwise the hyperparameters from the best-performing `Population` in experiment 2 were re-used. The values used are listed in Table 4.8 on page 58; we use logarithmic series of values again.

The highest-accuracy network from this experiment is shown in Figure 4.9 on page 61. We can see from Tables 4.7 on page 57 and 4.9 on page 60 that the population with the best accuracy in experiment 2 had similar performance in experiment 3, with a difference in accuracy of roughly 0.005 or 0.5% between the two experiments.

Recall from Section 4.2.2 that the fittest population from experiment 1 also had similar performance when reappearing in experiment 2. More experimentation would be warranted before we could state so with confidence, but this does suggest that it is possible that we are working on a sufficiently large scale for the law of large numbers (Yao & Gao, 2016) to push our algorithms towards consistent, repeatable performance.

Another notable finding from experiment 3 is that the top populations all had `mutation_probability` of 0.1, which, before this experiment, we had expected would prove to be too high. This opens up the possibility that we may want to explore higher values of `mutation_probability`.

One thing we do not see in experiment 3 is an improvement in maximum accuracy over experiment 2. After experiment 1, we decided to hold `min_n_genes` and `max_n_genes` constant for all populations in the grid search. In the next experiment, we will once again let them vary.

4.2.4 Experiment 4

As stated above, our primary intent in this experiment was to investigate the effects of letting the bounds on the gene length vary. We are also interested in observing the interactions between the bounds and `elitism_fraction`. Hyperparameters for this experiment appear in Table 4.10 on page 60.

The results of the experiment are in Table 4.11 on page 62. Unfortunately, this experiment shows no significant improvement in accuracy over previous experiments, nor is any connection between the varied hyperparameters and accuracy apparent.

The best individual of this experiment is too large and complex to include a diagram without shrinking it into illegibility. That individual consisted of 93 blocks in 8 layers, of which none were average-pool or sum blocks, 3 were max-pool blocks,

Maximum accuracy	Minimum # genes	Maximum # genes	Elitism fraction	Mutation probability	Mean threshold	Standard deviation threshold
0.681	10	20	0.4	0.1	0.01	0.001
0.680	10	20	0.2	0.1	0.01	0.001
0.678	10	20	0.05	0.1	0.01	0.001
0.672	10	20	0.8	0.1	0.01	0.001
0.669	10	20	0.1	0.1	0.01	0.001
0.668	10	20	0.1	0.01	0.01	0.001
0.656	10	20	0.2	0.01	0.01	0.001
0.650	10	20	0.2	0.001	0.01	0.001
0.650	10	20	0.05	0.01	0.01	0.001
0.646	10	20	0.4	0.001	0.01	0.001
0.636	10	20	0.1	0.001	0.01	0.001
0.630	10	20	0.05	0.001	0.01	0.001
0.615	10	20	0.8	0.001	0.01	0.001
0.610	10	20	0.8	0.01	0.01	0.001
0.588	10	20	0.4	0.01	0.01	0.001

Table 4.9: Performance of evolved CIFAR-10 classifiers, experiment 3

Hyperparameter(s)	Possible values
(Minimum number of genes, maximum number of genes)	(10, 20), (20, 30), (30, 40)
Elitism fraction	0.05, 0.1, 0.2, 0.4, 0.8
Mutation probability	0.1
(Mean threshold, standard deviation threshold)	(0.01, 0.001)

Table 4.10: Hyperparameters varied by grid search in CIFAR-10 experiment 4

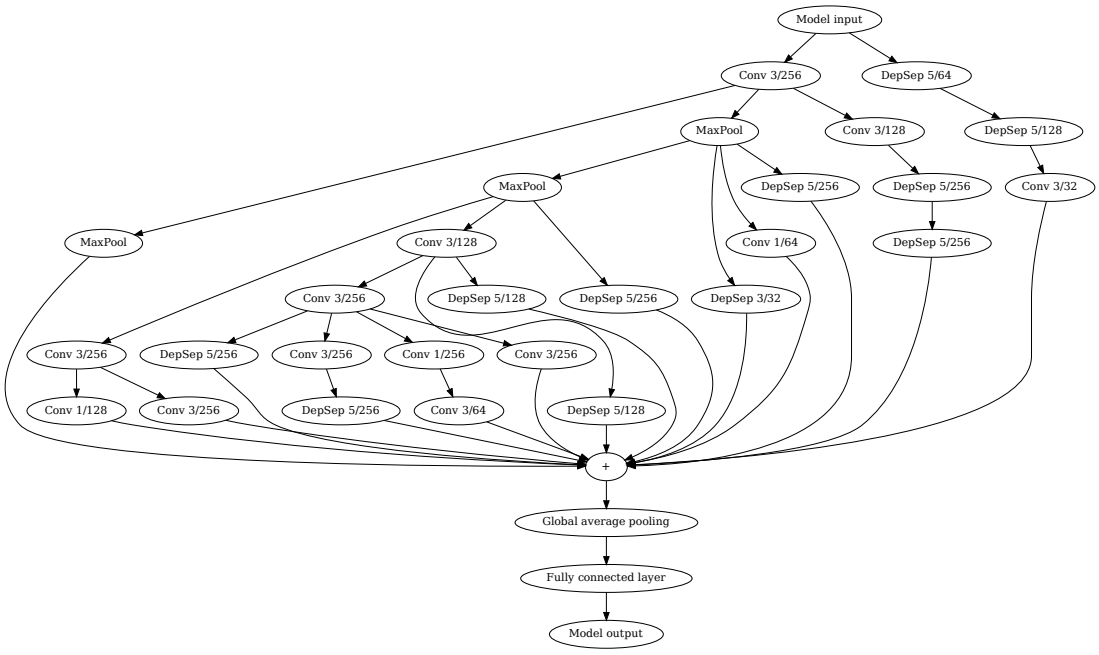

Figure 4.9: Topology of best-performing network from experiment 3. "Conv n/d" indicates a convolutional block with an n by n kernel and d output feature maps. "DepSep n/d" indicates likewise for depthwise separable convolutional blocks.

Maximum accuracy	Minimum # genes	Maximum # genes	Elitism fraction	Mutation probability	Mean threshold	Standard deviation threshold
0.688	30	40	0.8	0.1	0.01	0.001
0.688	30	40	0.05	0.1	0.01	0.001
0.687	30	40	0.2	0.1	0.01	0.001
0.686	10	20	0.05	0.1	0.01	0.001
0.686	20	30	0.4	0.1	0.01	0.001
0.684	30	40	0.4	0.1	0.01	0.001
0.683	10	20	0.4	0.1	0.01	0.001
0.681	30	40	0.1	0.1	0.01	0.001
0.677	10	20	0.1	0.1	0.01	0.001
0.676	20	30	0.8	0.1	0.01	0.001
0.675	20	30	0.1	0.1	0.01	0.001
0.671	20	30	0.2	0.1	0.01	0.001
0.671	20	30	0.05	0.1	0.01	0.001
0.665	10	20	0.2	0.1	0.01	0.001
0.642	10	20	0.8	0.1	0.01	0.001

Table 4.11: Performance of evolved CIFAR-10 classifiers, experiment 4

5 were concatenation blocks, 9 were convolution blocks, and the remaining 76 were depthwise seperable convolution blocks.

4.2.5 Experiment 5

At this point, we were somewhat disappointed with the accuracy of the classifiers we had evolved. While all performed well above the chance level of 10% accuracy, the CIFAR-10 webpage (Krizhevsky, 2009a) reports a reference CNN implementation with "18% test error without [data] augmentation", hence 82% accuracy, better than our best result so far of 68.8%. The current state of the art in CIFAR-10 classifiers is at least 97% accuracy (Kolesnikov et al., 2020).

Recall that, in all the experiments above, we used only a subset of the CIFAR-10 training data for training purposes–4,500 out of 50,000 examples, or 9%. In this

experiment, we wanted to examine the results of taking CNNs evolved using this cut-down training dataset, but training them for the purposes of evaluating their final accuracy using full training data.

We took the `Genome`s with highest accuracy from each of the previous experiments, and for each of those four `Genome`s, we once again converted it to an `Individual`, trained it for 50 epochs, and recorded accuracy on the test dataset after every 5^{th} epoch. This time, however, each epoch of training presented the entire 50,000-image training dataset, rather than the 4,500-image subset. In this experiment, the maximum recorded accuracy for each of the representatives of the previous experiments was in each case slightly higher: the CNNs from experiments 1 through 4 had maximum accuracy of 0.7627, 0.8322, 0.8033, and 0.7943 respectively. Note that the best network of experiment 2 (depicted in Figure 4.8 on page 58) now performs slightly better than the aforementioned reference implementation, 83.22% versus 82%.

4.3. Testing Different Variations on the Base Algorithm

In this series of experiments, we attempt different variations on the basic SDAG algorithm, in the hopes of improving on the best accuracies found in Section 4.2. For clarity, we refer to the different variations on the base SDAG algorithm as "SDAG-A", "SDAG-B", and so on through "SDAG-F".

4.3.1 Experiment 6

In this experiment, we explored the effects of restricting the "building blocks" available to the algorithm. We created an algorithm SDAG-A, identical to SDAG except for these changes:

- The only kinds of blocks used are convolution, depthwise separable convolution,

Hyperparameter(s)	Possible values
Elitism fraction	0.05, 0.1, 0.2, 0.4
Mutation probability	0.05, 0.1
(Mean threshold, standard deviation threshold)	(0.01, 0.001)

Table 4.12: Hyperparameters varied by grid search in CIFAR-10 experiment 6

and max-pooling.

- All convolution and depthwise separable convolution blocks have kernel size 3 (and consequently, no kernel-size-change mutations are used).
- Deletion mutations are not used.
- The hyperparameters `min_n_genes` and `max_n_genes` are not used. Instead, all genomes created in the initial population have exactly one (randomly generated) gene.

Hyperparameters for this experiment appear in Table 4.12 on page 64.

The results of the experiment are in Table 4.13 on page 66. What surprised us in these results is that this stripped-down version of SDAG produced three separate results with accuracy within 0.5% of 68.8%, our best accuracy from using the full SDAG algorithm in Section 4.2. The highest-accuracy network from this experiment is shown in Figure 4.10 on page 65.

4.3.2 Experiment 7

The algorithm for this experiment, SDAG-B, is identical to SDAG-A from Section 4.3.1, except that sum blocks are once again used. The hyperparameters were as in experiment 6, shown in Table 4.12 on page 64.

The results of the experiment are in Table 4.14 on page 66. Once again, the best results from this simplified version of SDAG are comparable to, and in one case

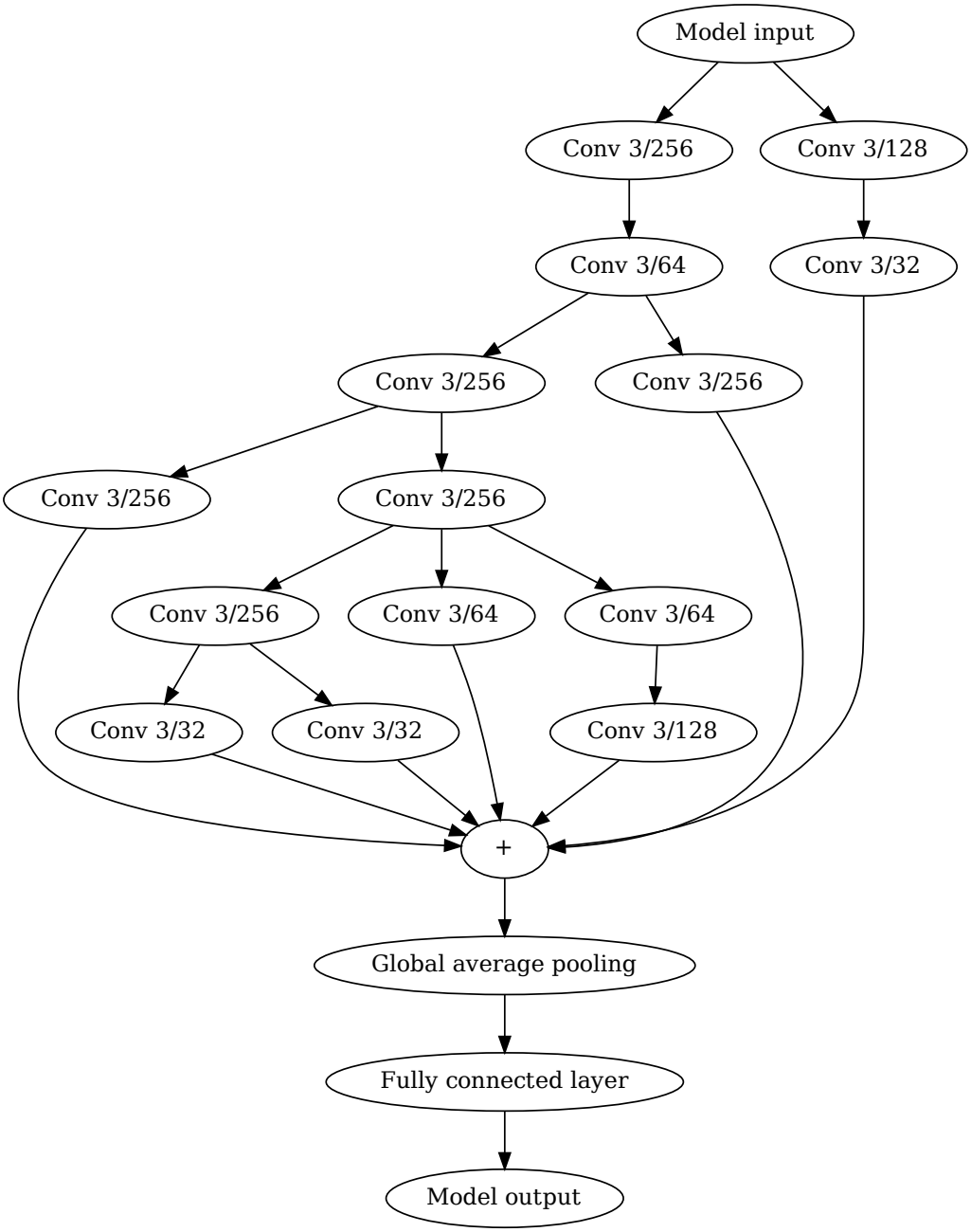

Figure 4.10: Topology of best-performing network from experiment 6. "Conv n/d" indicates a convolutional block with an n by n kernel and d output feature maps. "DepSep n/d" indicates likewise for depthwise separable convolutional blocks.

Maximum accuracy	Minimum # genes	Maximum # genes	Elitism fraction	Mutation probability	Mean threshold	Standard deviation threshold
0.687	1	1	0.4	0.1	0.01	0.001
0.686	1	1	0.05	0.05	0.01	0.001
0.684	1	1	0.1	0.1	0.01	0.001
0.669	1	1	0.1	0.05	0.01	0.001
0.667	1	1	0.4	0.05	0.01	0.001
0.656	1	1	0.05	0.1	0.01	0.001
0.648	1	1	0.2	0.05	0.01	0.001
0.642	1	1	0.2	0.1	0.01	0.001

Table 4.13: Performance of evolved CIFAR-10 classifiers, experiment 6

Maximum accuracy	Minimum # genes	Maximum # genes	Elitism fraction	Mutation probability	Mean threshold	Standard deviation threshold
0.690	1	1	0.2	0.1	0.01	0.001
0.689	1	1	0.2	0.05	0.01	0.001
0.687	1	1	0.1	0.1	0.01	0.001
0.669	1	1	0.4	0.05	0.01	0.001
0.656	1	1	0.05	0.05	0.01	0.001
0.652	1	1	0.1	0.05	0.01	0.001
0.641	1	1	0.4	0.1	0.01	0.001
0.638	1	1	0.05	0.1	0.01	0.001

Table 4.14: Performance of evolved CIFAR-10 classifiers, experiment 7

exceed, the 68.8% accuracy record set in Section 4.3. The highest-accuracy network from this experiment is shown in Figure 4.11 on page 67. Examining both Figure 4.10 and 4.11, we note that the best CNNs in both runs use only convolution layers, apart from the fixed tail.

4.3.3 Experiment 8

SDAG-C, the algorithm variation for this experiment, differs from SDAG-B from Section 4.3.2 in that, when converting a genome to an individual, the "tail"

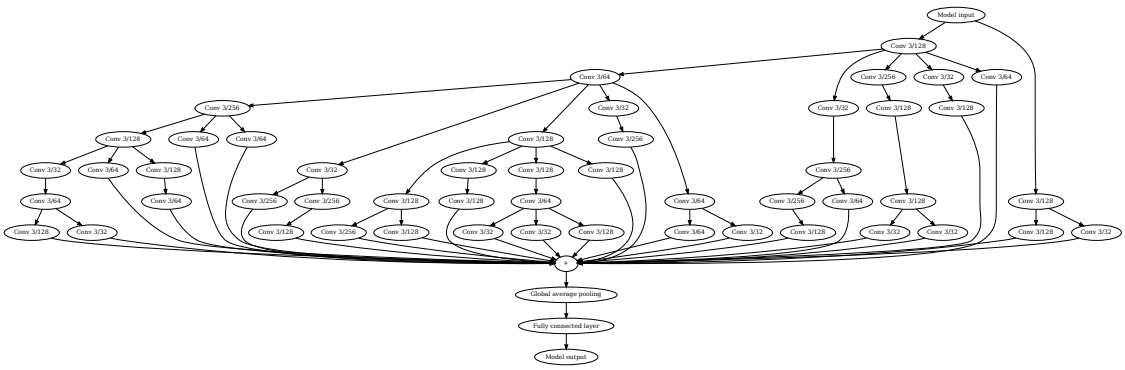

Figure 4.11: Topology of best-performing network from experiment 7. "Conv n/d" indicates a convolutional block with an n by n kernel and d output feature maps. "DepSep n/d" indicates likewise for depthwise separable convolutional blocks.

Hyperparameter(s)	Possible values
Elitism fraction	0.05, 0.1, 0.2, 0.4
Mutation probability	0.05, 0.1, 0.2, 0.4
(Mean threshold, standard deviation threshold)	(0.01, 0.001)

Table 4.15: Hyperparameters varied by grid search in CIFAR-10 experiment 8

referred to in Section 3.2.1 uses no global average-pooling layer.

Hyperparameters for this experiment appear in Table 4.15 on page 67.

The results of the experiment are in Table 4.16 on page 68. All results from this run were very poor: the best recorded accuracy is 36.1%, or just over half of current overall best accuracy of 69%, from Section 4.3.2. The highest-accuracy network from this experiment (such as it is) is shown in Figure 4.12 on page 69. Notably, this network consists only of max-pooling layers.

4.3.4 Experiment 9

The basis of SDAG-D is once again the full "vanilla" version of SDAG, but like SDAG-C in Section 4.3.3, no global average-pooling layer is used. Otherwise SDAG-D is the same as SDAG. Hyperparameters for this experiment appear in Table 4.17 on page 68.

Maximum accuracy	Minimum # genes	Maximum # genes	Elitism fraction	Mutation probability	Mean threshold	Standard deviation threshold
0.361	1	1	0.4	0.1	0.01	0.001
0.345	1	1	0.1	0.1	0.01	0.001
0.344	1	1	0.1	0.2	0.01	0.001
0.344	1	1	0.1	0.05	0.01	0.001
0.344	1	1	0.2	0.1	0.01	0.001
0.344	1	1	0.05	0.4	0.01	0.001
0.343	1	1	0.2	0.4	0.01	0.001
0.343	1	1	0.1	0.4	0.01	0.001
0.343	1	1	0.05	0.2	0.01	0.001
0.343	1	1	0.2	0.05	0.01	0.001
0.343	1	1	0.05	0.05	0.01	0.001
0.343	1	1	0.4	0.2	0.01	0.001
0.342	1	1	0.2	0.2	0.01	0.001
0.341	1	1	0.4	0.4	0.01	0.001
0.337	1	1	0.4	0.05	0.01	0.001
0.336	1	1	0.05	0.1	0.01	0.001

Table 4.16: Performance of evolved CIFAR-10 classifiers, experiment 8

Hyperparameter(s)	Possible values
(Minimum number of genes, maximum number of genes)	(10, 20)
Elitism fraction	0.05, 0.1, 0.2, 0.4
Mutation probability	0.05, 0.1, 0.2, 0.4
(Mean threshold, standard deviation threshold)	(0.01, 0.001)

Table 4.17: Hyperparameters varied by grid search in CIFAR-10 experiment 9

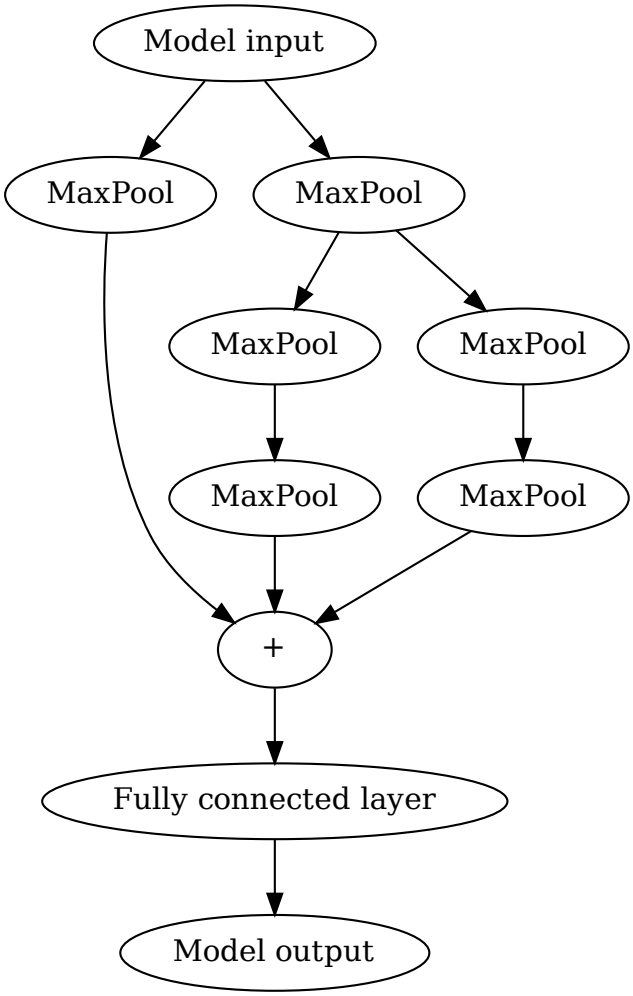

Figure 4.12: Topology of best-performing network from experiment 8. "Conv n/d" indicates a convolutional block with an n by n kernel and d output feature maps. "DepSep n/d" indicates likewise for depthwise separable convolutional blocks.

Maximum accuracy	Minimum # genes	Maximum # genes	Elitism fraction	Mutation probability	Mean threshold	Standard deviation threshold
0.637	10	20	0.2	0.05	0.01	0.001
0.609	10	20	0.4	0.05	0.01	0.001
0.607	10	20	0.4	0.2	0.01	0.001
0.594	10	20	0.05	0.05	0.01	0.001
0.593	10	20	0.1	0.05	0.01	0.001
0.589	10	20	0.1	0.2	0.01	0.001
0.588	10	20	0.4	0.1	0.01	0.001
0.580	10	20	0.1	0.1	0.01	0.001
0.570	10	20	0.05	0.2	0.01	0.001
0.545	10	20	0.2	0.1	0.01	0.001
0.515	10	20	0.05	0.1	0.01	0.001
0.479	10	20	0.2	0.2	0.01	0.001
0.382	10	20	0.05	0.4	0.01	0.001
0.382	10	20	0.1	0.4	0.01	0.001
0.380	10	20	0.2	0.4	0.01	0.001
0.356	10	20	0.4	0.4	0.01	0.001

Table 4.18: Performance of evolved CIFAR-10 classifiers, experiment 9

The results of the experiment are in Table 4.18 on page 70. Results here were far better than in Section 4.3.3, but also are slightly inferior to the results from all but one of the previous experiments.

The highest-accuracy network from this experiment is shown in Figure 4.13 on page 71. Note that both of the possible paths from the input node to the output node run through several average-pool layers. The shorter one leads through four average-pool layers, each of which reduces the height and width by a factor of two. Since CIFAR-10 images are 32 pixels by 32 pixels, passing an image through four average-pooling layers results in a tensor with height and width of $\frac{32}{2^4} = 2$ pixels. We could say that, although we eliminated the global average-pooling layer, an approximation of one has evolved.

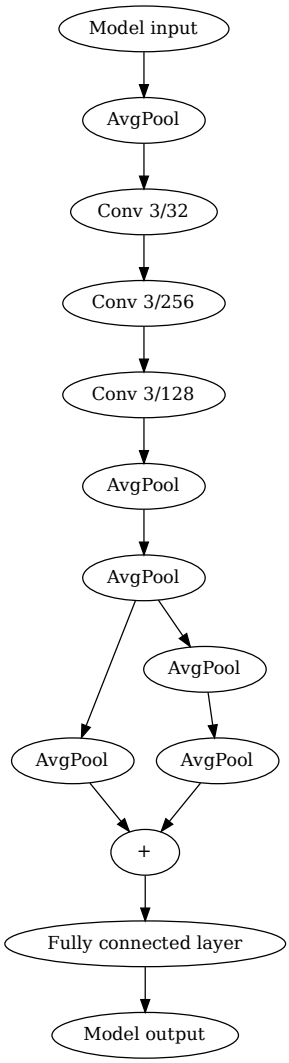

Figure 4.13: Topology of best-performing network from experiment 9. "Conv n/d" indicates a convolutional block with an n by n kernel and d output feature maps. "DepSep n/d" indicates likewise for depthwise separable convolutional blocks.

4.3.5 Experiment 10

To create SDAG-E, we began with SDAG-B from Section 4.3.2, and, inspired by Real et al.'s AmoebaNet-A classifier (Real et al., 2019), modified our tournament algorithm to favor younger genomes. We call the new tournament method "oversampled ageist slack binary tournament selection".

We define the age of all genomes in the initial generation to be 0. When breeding a new generation of genomes from an old one, if a genome is selected for inclusion in the new generation by elitism, and it is not subjected to any mutations, its age increases by 1. A genome selected via elitism with mutations applied, or a genome created via crossover, has its age set to 0.

Whereas the slack binary tournament shown in Algorithm 3.9 on page 35 begins by choosing two arbitrary genomes from the current population, our variation here chooses three arbitrary genomes instead. It then discards the genome with the highest age (choosing randomly in case of a tie), and then proceeds as a normal slack binary tournament with the remaining two genomes.

We made three attempts at this experiment. The first attempt was as described above. In the second attempt, we added deletion mutations back in, set up so that insertion mutations were twice as likely as deletion mutations. The third attempt was the same, except with insertion mutations only 1.25 times as likely as deletion mutations.

Each attempt ended up the same way: the amount of wall-clock time needed to evaluate each generation kept rising over the course of each run, which is consistent with our experiences in earlier experiments. However, this effect was much more pronounced with SDAG-E, and each of these runs reached a point where we judged performance to be too slow to make continuing the experiment practical, defined as

Hyperparameter(s)	Possible values
Elitism fraction	0.05, 0.1, 0.2, 0.4
Mutation probability	0.05, 0.1, 0.2, 0.4
(Mean threshold, standard deviation threshold)	(0.01, 0.001)

Table 4.19: Hyperparameters varied by grid search in CIFAR-10 experiment 10

three consecutive generations each taking over 60 minutes. We did say in Section 3.2.2 that we were not overly concerned about performance in this project, but we also do not have an unlimited amount of time to run experiments. In Chapter 5 below, we discuss some ideas for future variations on SDAG-E to keep the run time manageable. There is also the fact that, despite the hardware upgrade mentioned in Section 4.2.2, the amount of computing power available to us is still quite modest by institutional standards. A researcher with access to a more powerful platform might find SDAG-E's performance to be satisfactory.

Hyperparameters for this experiment appear in Table 4.19 on page 73. The same hyperparameters were used for all three attempts.

4.3.6 Experiment 11

Our final variation, SDAG-F, adds the oversampled ageist slack binary tournament from SDAG-E in Section 4.3.5 to the base SDAG algorithm. We further varied the algorithm by introducing a new hyperparameter, `ageism_factor`, which should have a value between 0 and 1 inclusive. When conducting a tournament, we still choose three arbitrary genomes and discard the oldest as in SDAG-E, except that before comparing the ages of the genomes, we multiply each age by `ageism_factor` and round down.

For example, suppose we select three candidate genomes A, B, and C, with ages of 1, 2, and 4 respectively. If `ageism_factor` is 0.3, then A, B, and C have adjusted ages $\lfloor 1 * 0.3 \rfloor = 0$, $\lfloor 2 * 0.3 \rfloor = 0$, and $\lfloor 4 * 0.3 \rfloor = 1$ respectively. Hence,

Hyperparameter(s)	Possible values
(Minimum number of genes, maximum number of genes)	(10, 20)
Elitism fraction	0.2
Mutation probability	0.1
(Mean threshold, standard deviation threshold)	(0.01, 0.001)
Ageism factor	0.125, 0.25, 0.5

Table 4.20: Hyperparameters varied by grid search in CIFAR-10 experiment 11

Max accuracy	Ageism factor	Min # genes	Max # genes	Elitism fraction	Mutation prob	Mean threshold	Standard deviation threshold
0.690	0.5	10	20	0.2	0.1	0.01	0.001
0.685	0.25	10	20	0.2	0.1	0.01	0.001
0.678	0.125	10	20	0.2	0.1	0.01	0.001

Table 4.21: Performance of evolved CIFAR-10 classifiers, experiment 11

C would be discarded as the oldest. However, if `ageism_factor` is 0.2, then the adjusted ages are all 0, and so an arbitrary choice out of the three genomes would be discarded. `ageism_factor` thus gives us a way to adjust how much disadvantage older genomes face during selection, by effectively mapping the large range of possible ages to a smaller set of "stages of life". Note that an `ageism_factor` of 0 makes these ageist tournaments behave like regular slack binary tournaments, effectively reducing SDAG-F to regular SDAG. Conversely, `ageism_factor` of 1 means that the tournaments have the same behavior as those of SDAG-E.

Hyperparameters for this experiment appear in Table 4.20 on page 74. The results of the experiment are in Table 4.21 on page 74. Comparing with Table 4.14 on page 66, we see that the performance of the best individual in this experiment is identical, to three significant figures, to the best individual from experiment 7. However, the individual in experiment 7 achieves this performance with a fraction of the number of blocks as in this experiment.

As in experiment 4, the best individual of this experiment is too large and complex to reasonably represent with a diagram. That individual consisted of 131 blocks in 9 layers, of which none were max-pool, concatenation, or sum blocks; 34 were average-pool blocks; 10 were convolution blocks; and the remaining 87 were depthwise separable convolution blocks.

4.3.7 Experiment 12

For this final experiment, we repeated the procedure from experiment 5 in Section 4.2.5, using the best `Genome`s from experiments 6, 7, and 11 as described in Sections 4.3.1, 4.3.2, and 4.3.6 respectively. The maximum recorded accuracies for the three `Genome`s examined were, respectively, 0.7942, 0.8231, and 0.805. These are comparable, though slightly inferior, to the results of experiment 5.

Chapter V. Summary

In this project, we first tested the tendency hypothesis of Sun, Xue, Zhang, and Yen, as described in Section 2.5.1. The evidence from those experiments suggests that the tendency hypothesis does indeed hold, and that furthermore, when evaluating the tendency of an individual, as little as a single epoch of training before evaluating the individual on the validation dataset may suffice.

In the second series of experiments, we examined the effects of varying the hyperparameters of SDAG on the performance of evolved CIFAR-10 classifiers. In those experiments, we found that the slack thresholds and mutation probability have the most effect upon the performance of the resulting classifiers, and the other hyperparameters relatively little effect. We had hoped to evolve a classifier that, evolved and trained on 10% of the CIFAR-10 dataset, could generalize to the full CIFAR-10 test dataset with competitive accuracy. We did not succeed in that, although in Section 4.2.5, when we used the small dataset for evolution but the full dataset for training, we did get accuracy comparable to, and in one case exceeding, an established baseline algorithm's performance of 82% (Krizhevsky, 2009a). With the current state of the art in CIFAR-10 classification being at least 97% accuracy (Kolesnikov et al., 2020), we had hoped to find a classifier with at least 90% accuracy.

In the third series of experiments, we varied the SDAG algorithm itself. We produced six variants on SDAG, designated SDAG-A through SDAG-F.

SDAG-A and SDAG-B were experiments which removed many primitives from

SDAG. These simplified versions of SDAG produced results comparable to, and in one case improving on, the best results gotten so far by SDAG. The best networks from both of these experiments consisted only of convolution layers, suggesting that it might be worthwhile to try an even further simplified version of SDAG in which we cut all layer types except convolution layers, as further discussed in Section 5.1.

SDAG-C and SDAG-D were identical to SDAG-B and SDAG respectively, but with the global average-pool layer removed from the tail of evolved individuals. SDAG-C produced very poor results, and while SDAG-D produced much better results than SDAG-C, they were still inferior to the results of all but one other previous experiment. The best-performing individual also incorporated multiple average-poooling layers, essentially mostly evolving the global average-pool layer back. That, in conjunction with the poor results from SDAG-C, suggests to us that SDAG variations using a global average-pool layer will generally outperform equivalent variations without such a layer.

SDAG-E was a variation on SDAG-B that introduced an "ageism" mechanism which biased tournament selection in favor of younger genomes. We made three attempts to run an experiment with slightly different variations of SDAG-E, and we aborted each of those three attempts when the experiment slowed down to a degree that we found unacceptable.

SDAG-F was a variation on SDAG that incorporated an ageism mechanism similar to that in SDAG-E, but with a new hyperparameter that allowed tuning of the algorithm's bias towards younger genomes. While the best SDAG-F network tied with the best SDAG-B network for best accuracy of any experiment we ran in this project, the individual generated by SDAG-F was far more complex than that generated by SDAG-B.

5.1. Discussion

Two of the best-performing variations of SDAG were also two of the simplest, SDAG-A and SDAG-B in Sections 4.3.1 and 4.3.2 respectively. Although both of those versions had multiple types of blocks at their disposal, the best results from these two experiments used only convolution blocks. This suggests that it might be worth trying variations on SDAG that explicitly use only convolution blocks, or convolution blocks plus one or both of the binary-operator blocks. Another simplification of SDAG-A and SDAG-B is that only one kernel size (3×3) is used; one could undo this simplification and investigate the performance of algorithms like SDAG-A and SDAG-B but with multiple possible kernel sizes.

Although all of the `Block`s that we implemented in Chapter 3 are rather simple, in theory we could implement `Block`s that do calculations of arbitrary complexity. Many well-known CNN architectures such as ResNet (He et al., 2016), DenseNet (G. Huang et al., 2016), GoogLeNet (Szegedy et al., 2015), and Xception (Chollet, 2017) contain characteristic self-contained repeating motifs that we could capture in a `Block` class. This suggests an experiment in which we take one of the aforementioned network architectures or a similar one, implement `Block` classes for its characteristic subnetworks, and see if improved variations on that architecture can be evolved. Sun, Xue, Zhang, and Yen have already done some similar investigation (Sun, Xue, Zhang, & Yen, 2020b), although in that work they use a mix of blocks from different architectures.

Another form of evolutionary algorithm is "genetic programming", in which the individuals are not ANNs, but rather computer programs represented as abstract syntax trees (ASTs) (J. R. Koza, 1992). J. Koza et al. described an enhancement to the basic genetic programming algorithm in 1999, in which a subtree of an evolved

AST can be extracted and turned into a re-usable function, which that AST or another may call multiple times (J. Koza et al., 1999). This suggests to us a similar enhancement to SDAG, in which we can extract a subgraph of a CNN for re-use.

The individuals in genetic programming are trees, so given any node in one, there is a single well-defined subtree rooted at that node. Choosing a subtree to extract as a function is thus as simple as picking an arbitrary node. In the case of a DAG evolved by SDAG, however, nodes may have more than one edge leading to them, and the best method of picking a subgraph to extract is an open and more complex question. Alternatively, given the surprisingly good performance of SDAG-A in Section 4.3.1, we could use that variation as the base: since all block types used in SDAG-A have arity 1, any DAGs it generates will in fact be trees, and thus we could pick an arbitrary node and use the subtree rooted at that node after all.

Z. Huang et al. note that ResNet (He et al., 2016) made use of two techniques to make training of very deep CNNs possible: batch normalization, and shortcut connections (Z. Huang et al., 2020). As stated in Sections 2.2.1 and 2.2.2, SDAG's convolution and depthwise separable convolution apply batch normalization. Shortcut connections can also evolve in SDAG. Figure 5.1 shows a typical shortcut connection, in which node A is connected indirectly to node C via node B, but also directly by a separate connection–the shortcut connection. Note that, for SDAG to be able to evolve this structure, node C must have arity 2, as described in Section 2.2.5.

Those authors claim that they have made alterations to ResNet that allow "60%-80% training and inference speedup", as well as a reduction in error rate of up to 4.5% (Z. Huang et al., 2020). They did so by removing batch normalization and shortcut connections, and replacing the ReLU activation function of convolution layers with the scaled exponential linear unit (SELU) activation function, defined:

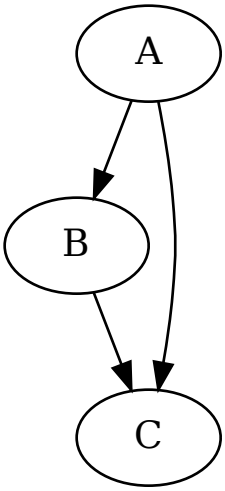

Figure 5.1: Typical shortcut connection

$$\varphi(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha\ e^x - \alpha & \text{if } x \leq 0 \end{cases} \tag{5.1}$$

with $\alpha \approx 1.6733$ and $\lambda \approx 1.0507$ (Z. Huang et al., 2020).

Although this work by Z. Huang et al. is in the domain of speech recognition, rather than image classification, the base ResNet algorithm that they modify was developed originally for computer vision problems, and was only later found useful for acoustic problems (Z. Huang et al., 2020). It therefore seems possible that their alterations might prove useful if re-imported to the visual world.

With shortcut connections removed, the architecture of ResNet is simply a chain of layers, with each layer taking exactly one input and having exactly one output (Z. Huang et al., 2020). A variation of SDAG that integrated Z. Huang et al.'s innovations could thus dispense with keeping track of the input indices of each

`Gene` and `Block`, and likewise the output indices of `Genome` and `Individual`. Instead, the order of `Gene`s within the `Genome` could imply the inputs and outputs.

References

Aggarwal, C. C. (2018). *Neural Networks and Deep Learning: A Textbook*. Springer International Publishing.

Angeline, P., Saunders, G., & Pollack, J. (1994). An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, *5*(1), 54–65.

Ashlock, D. (2006). *Evolutionary computation for modeling and optimization*. Springer.

Baccouche, M., Mamalet, F., Wolf, C., Garcia, C., & Baskurt, A. (2011). Sequential Deep Learning for Human Action Recognition. In A. A. Salah & B. Lepri (Eds.). D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, & G. Weikum (Eds.), *Human Behavior Unterstanding* (pp. 29–39). Springer Berlin Heidelberg.

Bolz, J., Farmer, I., Grinspun, E., & Schröder, P. (2003). Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics*, *22*(3), 917–924.

Bowler, P. J. (2003). *Evolution: The history of an idea* (3rd). University of California Press.

Bridle, J. S. (1990). Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition. In F. F. Soulié & J. Hérault (Eds.), *Neurocomputing* (pp. 227–236). Springer Berlin Heidelberg.

Caamaño, P., Bellas, F., & Duro, R. J. (2015). τ-NEAT: Initial experiments in precise temporal processing through neuroevolution. *Neurocomputing*, *150*, 43–49.

Chellapilla, K., Puri, S., & Simard, P. (2006, October). High performance convolutional neural networks for document processing. In G. Lorette (Ed.), *Tenth international workshop on frontiers in handwriting recognition*. Suvisoft.

Chen, L., & Alahakoon, D. (2006). NeuroEvolution of Augmenting Topologies with Learning for Data Classification. *2006 International Conference on Information and Automation*, 367–371.

Chinchor, N. (1992). MUC-4 evaluation metrics. *Proceedings of the 4th Conference on Message Understanding*, 22–29.

Chollet, F. (2017). Xception: Deep Learning with Depthwise Separable Convolutions. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1800–1807.

D'Ambrosio, D. B., Gauci, J., & Stanley, K. O. (2014). HyperNEAT: The First Five Years. In T. Kowaliw, N. Bredeche, & R. Doursat (Eds.), *Growing Adaptive Machines: Combining Development and Learning in Artificial Neural Networks* (pp. 159–185). Springer.

Dasgupta, D., & McGregor, D. (1992). Designing application-specific neural networks using the structured genetic algorithm. *[Proceedings] COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*, 87–96.

De Jong, K., Fogel, D. B., & Schwefel, H.-P. (1997). A history of evolutionary computation. *Handbook of Evolutionary Computation* (1st). IOP Publishing Ltd.

Fahlman, S. E. (1990). The recurrent cascade-correlation architecture. *Proceedings of the 3rd International Conference on Neural Information Processing Systems*, 190–196.

Fogel, D. (1994). An introduction to simulated evolutionary optimization. *IEEE Transactions on Neural Networks*, *5*(1), 3–14.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, *36*(4), 193–202.

Fullmer, B., & Miikkulainen, R. (1991). Using marker-based genetic encoding of neural networks to evolve finite-state behaviour. In F. J. Varela & P. Bourgine (Eds.), *Toward a practice of autonomous systems: Proceedings of the first European conference on artificial life* (pp. 255–262). MIT Press.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1994). *Design patterns: Elements of reusable object-oriented software* (1st ed.). Addison-Wesley Professional.

Granadeiro, V., Pina, L., Duarte, J. P., Correia, J. R., & Leal, V. M. S. (2013). A general indirect representation for optimization of generative design systems by genetic algorithms: Application to a shape grammar-based design system. *Automation in Construction*, *35*, 374–382.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

Holland, J. H. (1992, April 29). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press.

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, *2*(5), 359–366.

Huang, G., Liu, Z., van der Maaten, L., & Weinberger, K. Q. (2016). Densely connected convolutional networks.

Huang, Z., Ng, T., Liu, L., Mason, H., Zhuang, X., & Liu, D. (2020). SNDCNN: Self-normalizing deep CNNs with scaled exponential linear units for speech recognition. *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 6854–6858.

Hubel, D. H., & Wiesel, T. N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of Physiology*, *160*(1), 106–154.

Hutter, F., Kotthoff, L., & Vanschoren, J. (Eds.). (2019). *Automated Machine Learning: Methods, Systems, Challenges*. Springer International Publishing.

Ince, T., Kiranyaz, S., Eren, L., Askar, M., & Gabbouj, M. (2016). Real-Time Motor Fault Detection by 1D Convolutional Neural Networks. *IEEE Transactions on Industrial Electronics*, *63*.

Irwin-Harris, W., Sun, Y., Xue, B., & Zhang, M. (2019). A Graph-Based Encoding for Evolutionary Convolutional Neural Network Architecture Design. *2019 IEEE Congress on Evolutionary Computation (CEC)*, 546–553.

Jaafra, Y., Luc Laurent, J., Deruyver, A., & Saber Naceur, M. (2019). Reinforcement learning for neural architecture search: A review. *Image and Vision Computing*, *89*, 57–66.

Ji, S., Xu, W., Yang, M., & Yu, K. (2013). 3D Convolutional Neural Networks for Human Action Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Juba, B., & Le, H. S. (2019). Precision-recall versus accuracy and the role of large data sets. *Proceedings of the AAAI Conference on Artificial Intelligence*, *33*(01), 4039–4048.

Kampfner, R. R., & Conrad, M. (1983). Computational modeling of evolutionary learning processes in the brain. *Bulletin of Mathematical Biology*, *45*(6), 931–968.

Kassahun, Y., & Sommer, G. (2005). Efficient reinforcement learning through evolutionary acquisition of neural topologies. *ESANN 2005, 13th European Symposium on Artificial Neural Networks, Bruges, Belgium, April 27-29, 2005, Proceedings*, 259–266.

Kingma, D. P., & Ba, J. (2017, January 29). *Adam: A Method for Stochastic Optimization*. arXiv: 1412.6980 [cs]. Retrieved August 22, 2021, from http://arxiv.org/abs/1412.6980

Kohl, N., & Miikkulainen, R. (2012). An Integrated Neuroevolutionary Approach to Reactive Control and High-Level Strategy. *IEEE Transactions on Evolutionary Computation*, *16*(4), 472–488.

Kolesnikov, A., Beyer, L., Zhai, X., Puigcerver, J., Yung, J., Gelly, S., & Houlsby, N. (2020). Big Transfer (BiT): General Visual Representation Learning. In A. Vedaldi, H. Bischof, T. Brox, & J.-M. Frahm (Eds.), *Computer Vision – ECCV 2020* (pp. 491–507). Springer International Publishing.

Koutník, J., Schmidhuber, J., & Gomez, F. (2014). Evolving deep unsupervised convolutional networks for vision-based reinforcement learning. *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, 541–548.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press.

Koza, J., Andre, D., Keane, M., & Bennett, F. (1999). *Genetic programming III: Darwinian invention and problem solving*. Morgan Kaufmann.

Krizhevsky, A. (2009a). *CIFAR-10 and CIFAR-100 datasets*. Retrieved August 31, 2021, from https://www.cs.toronto.edu/~kriz/cifar.html

Krizhevsky, A. (2009b, April 8). *Learning Multiple Layers of Features from Tiny Images*.

Krüger, J., & Westermann, R. (2003). Linear algebra operators for GPU implementation of numerical algorithms. *ACM SIGGRAPH 2003 Papers*, 908–916.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, *1*(4), 541–551.

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324.

Li, Z., Liu, F., Yang, W., Peng, S., & Zhou, J. (2021). A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects. *IEEE Transactions on Neural Networks and Learning Systems*, 1–21.

Lozano, J. A. (2002). An Introduction to Evolutionary Algorithms. In P. Larrañaga & J. A. Lozano (Eds.), *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation* (pp. 3–25). Springer US.

Miguel, C. G., da Silva, C. F., & Netto, M. L. (2008). Structural and Parametric Evolution of Continuous-Time Recurrent Neural Networks, 177–182.

Miller, G. F., Todd, P. M., & Hegde, S. U. (1989). Designing neural networks using genetic algorithms. In J. D. Schaffer (Ed.), *Proceedings of the 3rd international conference on genetic algorithms* (pp. 379–384). Morgan Kaufmann.

Nguyen, G., Dlugolinsky, S., Bobák, M., Tran, V., López García, Á., Heredia, I., Malík, P., & Hluchý, L. (2019). Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: A survey. *Artificial Intelligence Review*, *52*(1), 77–124.

Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable Parallel Programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue*, *6*(2), 40–53.

Pardoe, D., Ryoo, M., & Miikkulainen, R. (2005). Evolving neural network ensembles for control problems. *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, 1379–1384.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., ... Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems 32* (pp. 8024–8035). Curran Associates, Inc.

Rawat, W., & Wang, Z. (2017). Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review. *Neural Computation*, *29*(9), 2352–2449.

Real, E., Aggarwal, A., Huang, Y., & Le, Q. V. (2019). Regularized Evolution for Image Classifier Architecture Search. *Proceedings of the AAAI Conference on Artificial Intelligence*, *33*(01), 4780–4789.

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q. V., & Kurakin, A. (2017, August 6–11). Large-scale evolution of image classifiers. In D. Precup & Y. W. Teh (Eds.), *Proceedings of the 34th international conference on machine learning* (pp. 2902–2911). PMLR.

Rothlauf, F. (2006). *Representations for genetic and evolutionary algorithms* (2nd ed). Springer.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536.

Saxena, A., & Saad, A. (2006). Genetic Algorithms for Artificial Neural Net-based Condition Monitoring System Design for Rotating Mechanical Systems. In A. Abraham, B. de Baets, M. Köppen, & B. Nickolay (Eds.), *Applied Soft Computing Technologies: The Challenge of Complexity* (pp. 135–149). Springer-Verlag.

Schwarz, M. W., Cowan, W. B., & Beatty, J. C. (1987). An experimental comparison of RGB, YIQ, LAB, HSV, and opponent color models. *ACM Transactions on Graphics*, *6*(2), 123–158.

Siebel, N. T., & Sommer, G. (2008). Learning defect classifiers for visual inspection images by neuro-evolution using weakly labelled training data. *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, 3925–3931.

Solla, S. (1992). Capacity control in classifiers for pattern recognition. *Neural Networks for Signal Processing II Proceedings of the 1992 IEEE Workshop*, 255–266.

Stanley, K. O., D'Ambrosio, D. B., & Gauci, J. (2009). A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks. *Artificial Life*, *15*(2), 185–212.

Stanley, K. O., & Miikkulainen, R. (2002). Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, *10*(2), 99–127.

Steinkraus, D., Buck, I., & Simard, P. (2005). Using GPUs for machine learning algorithms. *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, 1115–1120 Vol. 2.

Suganuma, M., Shirakawa, S., & Nagao, T. (2017). A genetic programming approach to designing convolutional neural network architectures. *Proceedings of the Genetic and Evolutionary Computation Conference*, 497–504.

Sun, Y., Xue, B., Zhang, M., & Yen, G. G. (2020a). Evolving Deep Convolutional Neural Networks for Image Classification. *IEEE Transactions on Evolutionary Computation*, *24*(2), 394–407.

Sun, Y., Xue, B., Zhang, M., Yen, G. G., & Lv, J. (2020). Automatically Designing CNN Architectures Using the Genetic Algorithm for Image Classification. *IEEE Transactions on Cybernetics*, *50*(9), 3840–3854.

Sun, Y., Xue, B., Zhang, M., & Yen, G. G. (2020b). Completely Automated CNN Architecture Design Based on Blocks. *IEEE Transactions on Neural Networks and Learning Systems*, *31*(4), 1242–1254.

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

Tan, M., Hartley, M., Bister, M., & Deklerck, R. (2009). Automated feature selection in neuroevolution. *Evolutionary Intelligence*, *1*(4), 271–292.

van Rijsbergen, C. J. (1979). Information Retrieval.

Verbancsics, P., & Harguess, J. (2015). Image Classification Using Generative Neuro Evolution for Deep Learning. *2015 IEEE Winter Conference on Applications of Computer Vision*, 488–493.

Wang, G., Cheng, G., & Carr, T. R. (2013). The application of improved NeuroEvolution of Augmenting Topologies neural network in Marcellus Shale lithofacies prediction. *Computers & Geosciences*, *54*, 50–65.

Whiteson, S., Stone, P., Stanley, K. O., Miikkulainen, R., & Kohl, N. (2005). Automatic feature selection in neuroevolution. *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, 1225–1232.

Wymann, B., Dimitrakakis, C., Sumner, A., Espie, E., & Guionneau, C. (2014). TORCS, The Open Racing Car Simulator.

Xie, L., & Yuille, A. (2017). Genetic CNN. *2017 IEEE International Conference on Computer Vision (ICCV)*, 1388–1397.

Xin Yao. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, *87*(9), 1423–1447.

Yao, K., & Gao, J. (2016). Law of Large Numbers for Uncertain Random Variables. *IEEE Transactions on Fuzzy Systems*, *24*(3), 615–621.

Yu, F., & Koltun, V. (2016). Multi-scale context aggregation by dilated convolutions. *International Conference on Learning Representations (ICLR)*.

Yuan, G., Xue, B., & Zhang, M. (2020). A Graph-Based Approach to Automatic Convolutional Neural Network Construction for Image Classification. *2020 35th International Conference on Image and Vision Computing New Zealand (IVCNZ)*, 1–6.