University of Southampton

Faculty of Engineering and Physical Sciences

Electronics and Computer Science

Evolving Convolutional Neural Network Topologies for Image Recognition

by

Yaron Strauch

September 2019

Supervisor: Dr Joanna Grundy
Second Examiner: Dr Martin D Charlton

A dissertation submitted in partial fulfilment of the degree of MSc Artificial Intelligence

ABSTRACT

Convolutional Neural Networks (CNNs) are state-of-the-art algorithms for image recognition. To configure a CNN properly by hand, one requires deep domain knowledge, best practises, and trial and error. As deep learning progresses, manual configuration gets harder and automated ways such as genetic algorithms are required. We describe in-depth how CNN topologies can be evolved; to do so the "half pixel problem" that occurs during programmatic CNN creation and manipulation is established, analysed and solved. A new human-readable genome representation for topologies and a novel ancestry tree visualisation for genetic algorithms is used to deepen understanding of the algorithm. We rediscover common design patterns within the topologies found and see when and how the algorithm can recover from wrong assumptions in its initialisation logic. Regularisation and partial training is introduced, allowing speed-ups of up to 19% while maintaining result accuracy.

Acknowledgements

The author acknowledges the use of the IRIDIS High Performance Computing Facility, and associated support services at the University of Southampton, in the completion of this work.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are state-of-the-art machine learning algorithms mainly used for but not limited to image recognition. In contrast to other Neural Networks, their architecture joins a set of different layer types, most notably *convolutional* and *pooling layers*, to sequentially pipe and transform data with. Figure 1.1 visualises a simple CNN with five layers detecting a hand-written digit. Early convolutional layers detect small basic features such as edges or corners and add new *channels*. Pooling layers reduce the resolution and leave the number of channels untouched. Reducing the resolution allows the following convolutional layers to recombine the basic features that were detected before into more sophisticated bigger ones specific to the data set. Convolutional layers recognise features using *filters*, matrices of learnable weights, and pooling layers use a mean or max function within their *kernels*. Both, filters and kernels, are moved over the input as a rolling window one *stride* step at a time, emitting one scalar per position. The input data can be padded with zeroes to counter decreasing resolutions. Moving the feature maps over the input reuses parameters and reduces the number of weights (in comparison to other neural networks), allowing more deeper topologies and detecting features independent of their location. The feature recognition part of the CNN is followed by a classifier to emit the desired output, classically a *multi layer perceptron* (MLP, often also called fully-connected network) followed by a softmax classifier. [15, 16, 13, 14]

The filter matrices are classically found through machine learning technologies, namely back propagation and gradient descend [16, 15]. This however can only be done if the overall architecture of the CNN is already known: The number of layers, the layer types, the number and size of filters/kernels, stride, padding, pooling functions, and number of hidden layers and neurons on the MLP. These parameters can and historically have been set by hand using trial and error, resulting in a range of named CNNs such as LeNet

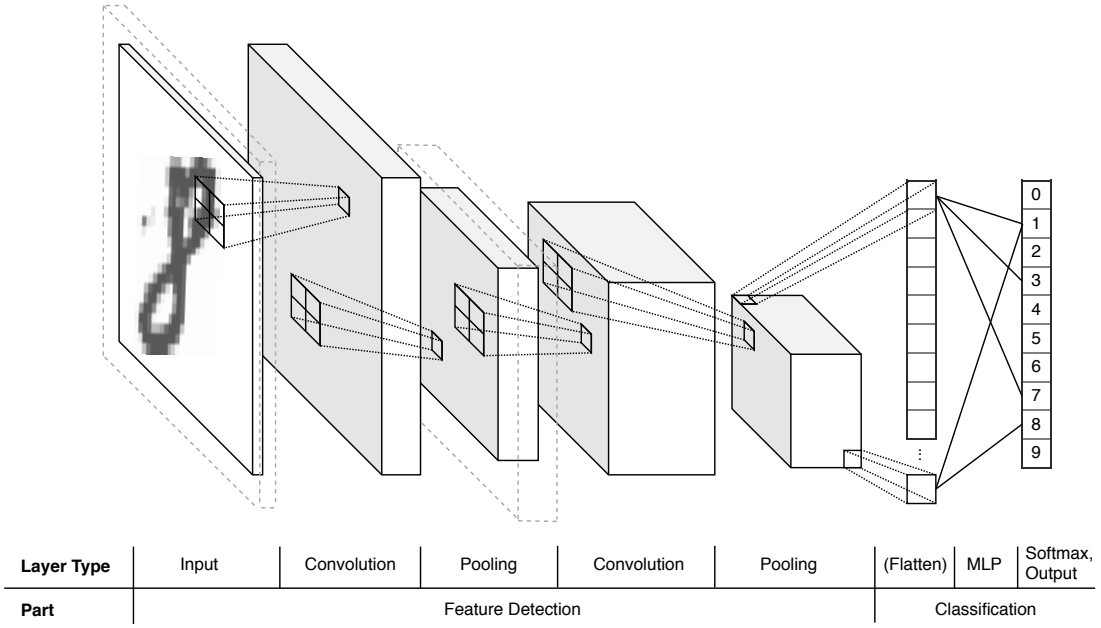


FIGURE 1.1: A visualisation of a CNN recognising hand-written digits. Convolutional layers detect features by multiplying learnable filter matrices with input data. They are moved over the data as a rolling window and emit a scalar into the next layer per location, adding one channel per filter (visualised by the data cube getting deeper). To ensure that convolutional layers do not reduce the width and height of the output data cube, the input cube is padded (grey dashed outline). Pooling layers resize the width and height of the data, but not the number of channels. In the end, a multi-layer perceptron and a softmax classifier emit the classification result. Based on [23].

[15], AlexNet [14], ResNet [9] and GoogleNet [27]. These architectures also introduced new features successively, namely dropout [14], modularity ("inception module", [27]), batch normalisation [11], and skip layers [9].

Skip layers consist of two to three convolutional layers and a skip connection that adds up the input of the first convolutional layer to the output of the last convolutional layer. This was shown to be useful for deepening layers and to mitigate the *vanishing gradient problem* [9].

All these parameters define a very big search space, and configuring is non-trivial and labour-intensive. Engineers can try out existing topologies and hope to achieve sufficient results. This still requires a lot of trial and error and a network that worked well on one data set is not guaranteed to work well on others. Or, they could design their own CNN topology, tailored to their data set, by following a range of best practises, i.e. [24, 11]. These conventions and best practises were published by domain experts, potentially introducing human bias for data sets outside of main research. Also, due to recent progress of deep learning, manual tuning of deep architectures becomes more complicated, non-intuitive and error-prone. Considering that deep networks train for hours or even days, tuning these parameters stretches out over a long period of time.

Because a CNN can have any number of layers and filters, the variety of CNNs is theoretically unlimited and exhaustive search is not an option. More systematic approaches to find good topologies and automatic ways to cut development cost are needed.

1.2 Genetic Algorithms

Genetic Algorithms [1] (GAs) (in some context also evolutionary algorithms [4]) are one of the approaches that could help with this task. They are heuristic convex optimisation algorithms designed to optimise within abstract high-parameter spaces where parameters have complex and unknown interdependencies. Their heuristic nature does not guarantee a global optimum or reproducible outcomes, but they have been shown to operate well on NP-hard problems [6], to get stuck less likely than local incremental search algorithms (*hill climbers*) [4], and to overcome human bias (i.e. [10]).

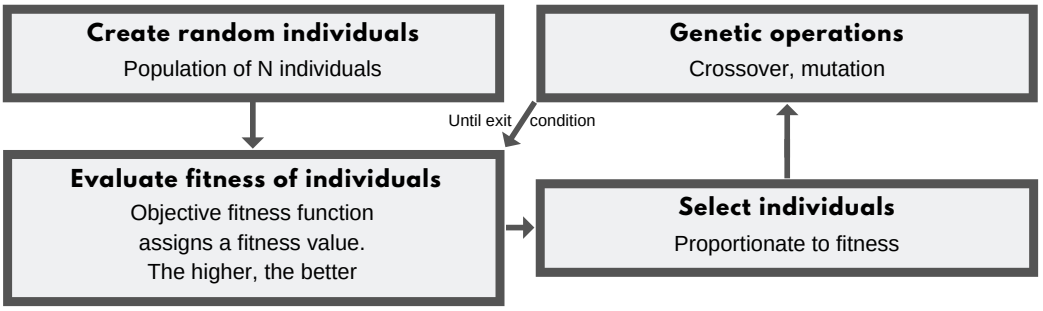


FIGURE 1.2: Standard GA flow

GAs implement a simplified model of evolution by natural selection, see figure 1.2. A population of individuals is initialised, where each individual is represented by a *genome*. Their *fitness* is measured by an *objective function* that maps each genome to a fitness score. Individuals are selected for reproduction according to their fitness, higher fitnesses are selected more likely. *Genetic operators* are used to manipulate their genome stochastically: *Mutation* is a small random change in the genome and *cross-over* recombines the genome of two parents, hopefully splicing together two good things and therefore causing a jump in fitness [18]. Individuals are either reproduced into a new generation of same size (*generational GA*) or into the same population, replacing a weaker individual (*steady state GA*). While the population as a whole converges towards an optimum due to the fitness-proportionate selection for reproduction, genetic operators are not directed towards an optimum and truly random. Reproduction, genetic operations and fitness evaluation are repeated until an exit condition occurs, classically after a set number of generations or when a fitness threshold is reached. [1]

1.3 Evolving CNNs using GAs

GAs have already been used to optimise CNNs in various ways.

Real et al. built a GA that evolves a population of CNNs [21] where the genome specifies both the CNN weights and a learning rate. Children inherit the weights of their parents if possible, mutation can insert, alter or remove layers, or change the learning rate. They train individuals successively from generation to generation using gradient descend and back propagation. The objective fitness function is measuring the accuracy of partially trained networks. This trick reduces training time on inferior individuals. The selection process does not need an absolute measure of fitness, it only needs to know if an individual has a higher or lower fitness relative to its generation.

A big drawback of this approach is that there is no cross-over involved. Introducing cross-over for weights is very complicated if not impossible because of the syntactical restrictions binding the output and input shape of adjacent layers to each other. Also there is a symmetry problem: If two CNNs are equal except for the order of its parameters, and the cross-over mechanism is not rearranging the parameters, crossing over these two genomes would result in a loss of information, rendering recombination useless. Such an ordering is not trivial to achieve.

The problem of the missing cross-over was overcome. Xie and Yuille built a genetic CNN [29] where the genome does not represent the weights of a CNN, but the overall topology. To do so they developed a binary CNN representation where mutation flips one bit at a time and cross-over recombines slices of the genome. The genome therefore describes the hyper parameter configuration, not the weights themselves. The trade-off of this methd is that in order for the objective fitness function to evaluate an individual, it needs to initialise a CNN according to the topology, train it using gradient-descend and back propagation, and test it. Unseen topologies, even if they are very similar to one observed before, need to be retrained all over again, requiring and potentially wasting computational resources.

The genome representation defined by Xie and Yuille encodes CNNs as inter-node connections in a bit string, requiring a healing algorithm to translate all possible genomes into valid CNN topologies. The resulting strings are not intuitive for humans to understand, and a small bit-flip mutation on the genome may result in big changes to the actual CNN layout or none at all, rendering the effect of a mutation hard to regulate.

The latest development of Sun et al. [26] focused on deepening the number of layers by extending Xie and Yuille's algorithm. They introduced Skip Layers and new probability distributions for the mutation operator, encouraging CNNs to become deeper. They did not specify any string representation of their genome, instead they used a linked list of components, resulting in no documentation of topologies found.

All of these papers found novelty algorithms that can compete with state-of-the-art networks. Xie and Yuille showed that their GA performs "much more efficient than random search in the large solution space" [29], strengthening the application of GAs for this task.

One of the main challenges of this algorithm is *wall time*: Xie and Yuille report they evolved for 17, Sun et al. for 35 *GPU days*. These durations result from the need of retraining unseen topologies from scratch and is one of the main limiting factors for the algorithm to be of real-world application.

1.4 Contribution

This work focuses on analysing, documenting, and speeding up a genetic algorithm that evolves CNN topologies for image detection.

More specificity, it will be investigated:

1. the "half pixel problem" and how to mitigate it
2. three variants of creating random CNN topologies for the initial generation and which one to use
3. if and how the algorithm adapts to the dataset CIFAR10 to MNIST
4. if reintroducing partial training and adding regularisation can reduce wall time without a loss in performance

To reach these goals, we will further:

1. define a human readable genome representation
2. introduce a GA visualisation of ancestry trees to understand the genetic processes

Chapter 2 will lay out a foundation for this algorithm, explaining the basics of all CNN and genetic components used throughout this work and point to appropriate literature for deeper research.

In chapter 3 the process of evolving CNN topologies will be laid out. The "half pixel problem" will be explained and analysed in section 3.1.1, and three approaches to mitigate this problem during CNN creation and manipulation will be established. Section 3.2 discusses how machine learning is used to train and test our topologies. The evolutionary process, including genome representations and visualisations, is described in section 3.3.

The algorithms are then tested within the six experiments documented in chapter 4. A base experiment that allows us to compare the core algorithm to relevant literature will be analysed. The base experiment will be extended with one modification at a time, allowing us to make comparisons to the base experiment and see if the modifications are successful.

Finally, chapter 5 will summarise our core results, pinpoint possible weaknesses, and document ideas for further research on this algorithm.

Chapter 2

Background

2.1 CNN Components

We will describe CNNs as a set of different components. We describe them in a bottom-up hierarchy, starting with the smallest component, until reaching the CNN itself as the topmost level.

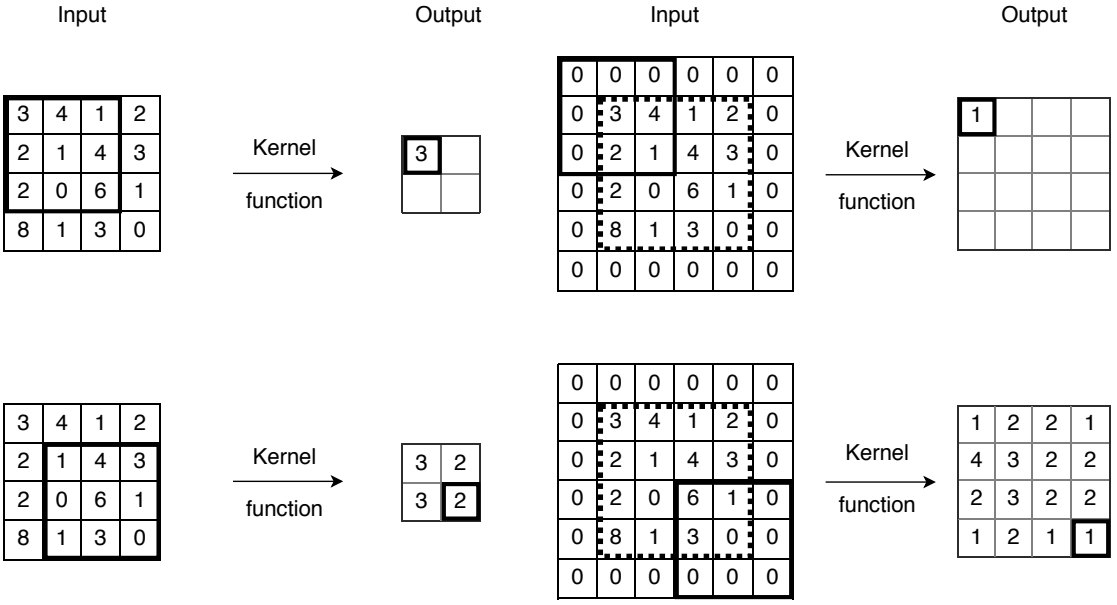
2.1.1 Kernels and Filters

Kernels and filters are functions that produce a scalar output for two dimensional input data. They are "moved" over an input matrix; for each location, the filter/kernel is applied for only a sub set of the input matrix and one scalar is emitted. The scalars are joined into an output matrix. The distance kernels and filters are moved each time is called *stride*. If the stride is 1x1, we also call it *unit stride*, and a kernel or filter of 1x1 is called *unit kernel* or *unit filter*.

The difference between filters and kernels is the function that transforms the input matrix into a scalar. Filters are matrices of learnable weights that produce a scalar using matrix multiplication, kernels use an average or max function on their input. Each filter adds one channel to the output; you can therefore have more than one filter per convolutional layer. Kernels do not change the number of channels.

From now on, we we will use the term "kernel" for both, filters and kernels, to increase readability and only specify "filter" if a distinction is needed.

Figure 2.1a illustrates how a 3x3 kernel is applied to a 4x4 input. It is moved one pixel at a time and emits the rounded average of the current area into the output matrix. After the kernel has been moved four times, the 2x2 output matrix has been completed. Instead of the average function, the kernel could also have applied a max function or multiplied it with a 3x3 feature map.



(A) Without padding, the output resolution of 2x2 is smaller than the input resolution.

(B) Same padding: By adding a 1x1 zero padding, the output resolution is equal to the input resolution. The dotted line shows the original input.

FIGURE 2.1: Visualisation of a 3x3 kernel (black rectangle) being applied at the first and last position of a 4x4 input matrix. For each position, the kernel emits a scalar that is joined into an output matrix. In this case, the kernel is moved by the unit stride and applies a rounded average function.

Note how the output size of 2x2 is smaller than the input size. This is the case for all kernels bigger than 1x1. To mitigate this, one can pad the input picture, classically with zeroes. A padding that achieves the same output resolution as the input resolution is called *same padding*, all other paddings will be referred to as *valid padding*.

In figure 2.1b, the same image is padded with one row / column of zeroes on each side (1x1 zero padding). This renders the output matrix to have the same resolution as the input. As a side note: This is only an example, normally same padding would not be used together with a mean/max kernel; kernels in pooling layers are supposed to reduce the resolution. But same padding is often used for the filters in a convolutional layer.

To calculate same padding, we will represent all parameters as scalars, assuming only squared resolutions. All equations can be transferred to the general case by using the variables as vectors instead. Let us represent the kernel size by K, the input and output resolution as I and O respectively, the stride as S, and *symmetric padding* as P. We call it symmetric padding because it is applied to both sides symmetrically, figure 2.1b shows $P = 1$.

If we use a unit kernel, a unit stride and no padding ($K = S = 1; P = 0$), then the output size will be equal to the input size. If we increase the kernel size, the output size will be decreased by the same amount, therefore $O \sim -K$. To counter this, we increase the padding. Because padding is applied on both sides $O \sim 2P$. Increasing the stride S

decreases O substantially, depending on I, P and K. Following [7], this results in (2.1) for the output size:

$$O = \frac{I + 2P - K}{S} + 1 \tag{2.1}$$

Notice that for suboptimal parameter configurations, O might result in a fraction instead of a whole number. Because there are no fraction of pixels, the layer would be invalid.

To calculate same padding P_s, the case where $I = O$, we can transform this formula into

$$P_s = \frac{S(I-1) + K - I}{2} \tag{2.2}$$

2.1.2 Convolutional Layer

The convolutional layer extracts new features from its input. It has n filters (n is defined in the respective experiment) whose weights are learned using gradient descend and backpropagation [2].

In order to minimise the impact of inserting and removing layers to an existing layer stack, the convolutional layers in this implementation were designed to not change image resolutions ($O = I$, same padding and unit stride $S = 1$). Inserting or removing a convolutional layer might change the number of channels, but it will not change the image resolution. Semantically speaking, the convolutional layer is only supposed to detect new features and not to reduce the resolution.

From (2.2), we can see that $P_s = \frac{K-1}{2}$ for unit stride. Because symmetric padding needs to be a whole number in pixels, the kernel size K must be an odd number. A kernel of 1 would not detect any features, so $K = 2n + 1; n > 0$ for convolutional layers.

The experiments conducted use filters of size 3x3 with same padding and unit stride. This has been proven to be successful in the past, e.g. [25].

Following [26] and general CNN design practises [11], a rectifier activation function ($ReLu$) and a batch normalisation is performed after each convolutional layer.

2.1.3 Skip Layer

Following [26], a skip layer has been introduced as an independent component. A skip layer classically consists of 2-3 convolutional layers in serial and a skip connection [9]. The skip connection adds the initial input that the first convolutional layer receives to the output of the last convolutional layer. This is suspected to mitigate the loss of

gradients in deep networks and was shown experimentally to be effective, most notably in ResNet [9]. Following [26], we will use Skip Layers of depth 2.

Because the skip connection adds two matrices, their shapes must match. If the input shape of the skip layer does not match its output shape, an adapter is needed before the skip connection can add the two matrices. Similar to [26], this adapter is an implicit convolutional layer that takes the original input and maps it to the correct shape using a unit stride and unit kernel with the appropriate number of feature maps.

2.1.4 Pooling Layer

A pooling layer reduces the resolution of the input by setting a non-uniform stride. The number of channels is not manipulated and every channel is processed individually. The kernel of a pooling layer applies either an average or a max function on its input, which means that there are no weights required. Because the pooling layer is meant to reduce the resolution, its kernel does not use same padding or unit stride.

Following [26], we limit both kernel resolution and stride to 2x2 and set padding to 0. This makes handling resolutions easier: Inserting $K = S = 2$ and $P = 0$ into (2.1) we see that $O = \frac{I-2}{2} + 1 = \frac{I}{2}$, allowing the pooling layers to always halve the input resolution. Because we will use images of size 2^n, pooling layers will work as long as the input is bigger than one pixel. How to prevent pooling layers to produce half a pixel will be analysed in section 3.1.1.

2.1.5 Softmax MLP Classifier

The last component of a convolutional neural network is a classifier. [26] documents using a softmax classifier (i.e. [8]), but they specify that no MLP (multilayer perceptron) is used in order to limit the search space. This seems to break best practises of CNNs [24]. It was therefore decided to incorporate an MLP, but limit the number of possible configurations through their search space drastically.

A softmax classifier needs exactly as many neurons as the desired number of output labels. It will chose the neuron with the highest value. To achieve this number of neurons, a MLP connects the output of the last convolutional/pooling layer to the output layer. Additionally, the MLP might have hidden layers in between, as defined in the search space of the experiments.

2.1.6 Convolutional Neural Network

The Convolutional Network reuses all the components described above in a sequential layer architecture. The first part combines a number of skip, convolutional and pooling layers. The second part uses the Softmax MLP classifier to generate the output.

This is not the only way to build a CNN, but it is the most simple and popular method. There are also architectures with more parallel and modular topologies, i.e. [27], but these networks are harder to create automatically using GAs.

2.2 Genetic Algorithms

2.2.1 Genetic Operators

GAs can be roughly classified into *asexual* and *sexual*.

In asexual GAs, individuals only reproduce by copying their genome and changing it for a very small amount, which is called *mutation*. The *mutation rate* describes the likelihood of a mutation to occur. It should not be too small to stagnate convergence, and it should not be too big to re-roll the child without resembling its parent. [5].

Sexual GAs have an additional mechanism to recombine two individuals using *cross-over*. There are three common cross-over techniques: *Uniform cross-over* iterates over both parental genomes and chooses either gene at random. The other two variants split each parental genome at one (*one point cross-over*) or two ((*two point cross-over*)) locations and recombine the slices alternating (maintaining order). [1, 4, 5]

Literature suggests not using uniform crossover when there is *epistasis* (fitness inter-dependencies between genes); two-point cross-over is preferred for cyclic genomes [5]. Past GAs evolving CNN topologies used one-point cross-over [29, 26]. This makes sense because each layer depends on its predecessors and the genome is acyclic.

2.2.2 Selective Pressure

Selective pressure describes how likely it is that individuals with relatively low fitness get eliminated from the population [4]. If the selective pressure is very high, only the fittest individual will reproduce and the variety of individuals will decrease, rendering the GA to be an expensive hill climber. On the other hand, if the selective pressure is too small, the population will not converge to an optimum because mutations are classically undirected [5].

The selection process will thin out less fit individuals, moving the average fitness towards an optimum. This means that both the fitness spread (or variance in fitness) and the selective pressure decreases [28]. This can be undesirable because it slows down convergence rate and reduces variety that is generally needed for GAs [5].

Selective pressure can be controlled by the selection algorithm. Popular choices of selection algorithms, sorted by descending selective pressure, are fitness-proportionate selections ("roulette-wheel"), where twice as fit individuals are selected twice as often, rank-based algorithms where each distinct fitness is assigned a rank and selection is proportionate to said rating, and tournament selection where individuals are selected pairwise and the better one is selected [4, 5].

Selective pressure also depends on the population size. A population with one individual is a hill climber with highest possible pressure, and the bigger the population, the greater the fitness spread and the less aggressive the selection - for infinite populations, every individual can reproduce. [5]

Chapter 3

Method

3.1 Programmatic CNN Creation and Manipulation

3.1.1 The Half Pixel Problem

One of the main issues that occurs during the programmatic handling of CNN topologies will be called *half pixel problem*.

As explained in section 2.1.1, kernels with valid padding and non-uniform stride and kernel size reduce the resolution of the data. Because image resolutions have to be natural numbers, they cannot decrease infinitely.

Unfortunately, the maximum number of layers using these kernels is determined by both the data set and the search space: The data set defines the resolution of the image and the search space defines possible kernel sizes, strides, and paddings. Intuitively, bigger images can be reduced more often, but only if the kernel search spaces are defined accordingly. For example, if the kernel search space is configured to halve the input resolution, and the input resolution is arbitrarily big but an odd number, the half pixel problem still occurs.

To simplify this discussion, we will assume that image source resolutions are always 2^n and only pooling layers reduce the resolution, and they always halve it. Further we assume to only use pooling and convolutional layers, both occurring equally likely. All approaches can be adapted to more general cases except when explicitly stated.

Under these assumptions, the n^{th} pooling layer will reduce the resolution to 1 pixel, and the $(n+1)^{\text{th}}$ pooling layer would be syntactically faulty.

3.1.2 Creating a random CNN layer stack

The half pixel problem becomes imminent when creating a random CNN layer stack. Three approaches to solve this issue were identified.

3.1.2.1 Approach 1: Early stopping

The first approach, depicted in algorithm 1, is to stop appending layers completely by the time the algorithm would add an invalid layer (line 9).

Algorithm 1: Stop appending layers when invalid resolutions occur

Require: D {Desired CNN depth}
1: $L \leftarrow []$ {Layer stack}
2: **while** len(L) $< D$ **do**
3: **if** rand() $\leq .5$ **then**
4: $L' \leftarrow$ push(L, new POOL())
5: **else**
6: $L' \leftarrow$ push(L, new CONV())
7: **end if**
8: **if** L' has invalid output shape **then**
9: **return** L
10: **end if**
11: $L = L'$
12: **end while**
13: **return** L

Assuming we were to create deep networks (i.e. $D = 100$) with only pool and convolutional layers, both equally likely, in expectation the invalid pool layer would occur in depth $2(n+1)$, rendering the actual expected network depth to be $2(n+1)-1 = 2n+1$. For CIFAR10 (see section 3.2.1.2), images are $2^n = 32 = 2^5$ big, which results in an expected depth of 11 instead of 100 layers. This algorithm therefore restricts the depth substantially.

3.1.2.2 Approach 2: Only append valid layers

The second approach, algorithm 2, is to not stop appending layers completely, but to stop appending layers that would produce invalid resolutions only (in our case pooling layers). This allows to have the desired depth and prevents invalid topologies.

There are two disadvantages. First, the density of pooling layers would suddenly drop to 0% right after the last possible pooling layer was added. This means that in expectation, the first $2n$ layers have a 50/50 ratio of pooling layers and all other layers are purely convolutional. The second disadvantage is that all those convolutional layers in the second part would operate on 1x1 pixel data that cannot hold much information. A

Algorithm 2: Only append valid layers

Require: D {Desired CNN depth}
1: $L \leftarrow []$ {Layer stack}
2: **while** len(L) $< D$ **do**
3: **if** rand() $\leq .5$ **then**
4: $L' \leftarrow$ push(L, new POOL())
5: **else**
6: $L' \leftarrow$ push(L, new CONV())
7: **end if**
8: **if** L' has valid output shape **then**
9: $L = L'$
10: **end if**
11: **end while**
12: **return** L

variant of this approach is to manipulate line 8 and stop adding pooling layers when the output shape is smaller than a given threshold. This would mitigate the loss of information but introduce another hyper parameter.

3.1.2.3 Approach 3: Change probability distribution

The last approach, algorithm 3, extends either of the first two. To ensure deep networks and evenly distributed pooling layers, the probability distribution of the layer type has been changed. The other two approaches used a 50/50 chance for the two layer types. The third approach shifts this ratio when needed: Under the assumption that resolutions are 2^n and that pooling layers halve resolutions, the maximum number of pooling layers is the $\log_2$ of the input size I. We can therefore calculate the mean/expected number of pooling layers $\bar{n}$ (line 2), which is either $log_2(I)$ or half the number of layers (equivalent to a 50/50 ratio), whichever is smaller. This expected number of pooling layers $\bar{n}$ is transformed to the probability of adding a pooling layer p (line 3). When based on algorithm 2, the depth D is ensured, when based on algorithm 1 the depth D is an upper bound. Which one we choose should not matter much as both variants ensure much deeper CNNs than approach 1 and space out pooling layers evenly.

The big drawback of algorithm 3 is that it does not adapt well to relaxed assumptions: More general cases of kernel configuration search spaces will make calculating the probability distribution complicated if not impossible. For example, if the algorithm chose same/valid padding randomly, the output shape of each layer would be non-deterministic and the relation $\bar{n} \sim \log_2(I)$ does not hold any more.

Algorithm 3: Change probability distributions to space out pooling layers

Require: D {Desired CNN depth}
Require: $I = 2^n$ {Input resolution}
1: $L \leftarrow []$ {Layer stack}
2: $\bar{n} \leftarrow \min(\frac{D}{2}, \log_2(I))$ {Expected number of pooling layers}
3: $p \leftarrow \frac{1}{\bar{n}}$ {Probability of adding a pooling layer}
4: **while** len(L) $< D$ **do**
5: **if** rand() $\leq p$ **then**
6: $L' \leftarrow$ push(L, new POOL())
7: **else**
8: $L' \leftarrow$ push(L, new CONV())
9: **end if**
10: ... {Proceed like in algorithm 1 or 2 line 8}
11: **end while**
12: **return** L

Table 3.1: CNN initialisation approaches

Name	Strategy	Pro	Con
Approach 1	Stop appending layers completely if half pixels would occur	Adapts well to general cases	Shallow networks
Approach 2	Skips appending layers that would produce half pixels	Deep networks	No pooling at deeper layers, deep skip layers would operate on small resolutions
Approach 3	Change probability distribution for layer types	Deep networks, pooling layers are spaced out correctly	Works only if each layer type has a deterministic resolution outcome, does not maintain 50/50 chance

3.1.2.4 Choosing the right approach

For comparison, the three approaches are summarised in table 3.1. Which of these three approaches is best will be explored in the experiments section, namely section 4.1, 4.2 and 4.3 for approach 1-3 respectively.

It will be found out that approach 2 is not working at all due to the distribution of pooling layers, as already expected. Further, it will be shown that in early generations, approach 3 has much weaker individuals than approach 1, but it almost fully recovers due to cross-over and mutations. Approach 1 resulted in the highest fitness and the fitness distributions look more stable and less depending on random chance. The networks of approach 1 were shallow (as expected), but the accuracies of shallower networks were significantly higher. Our assumption that "deeper networks are better" did not hold,

therefore the reservations against algorithm 1 were unsubstantiated in the context of our search space.

Because algorithm 1 had the best results and adapts well to more general cases, this approach was used for all other experiments.

3.1.3 Random CNN creation

A random CNN is more than a random layer stack. It needs to have a classifier attached, to propagate the input and output shapes, and choose a number of layers at random.

This process is documented in algorithm 4. The function expects a lower and upper bound for the number of layers, and a set of possible layers; these parameters are defined in the search space of the experiment. It also expects the input shape of the data (channels x width x height) and the number of labels for the classification task; both of these parameters are defined by the data set. It propagates the output shape of each layer to the following layer and adds a classifier in the end. This is important because each component needs to know its input shape in order to set up, i.e. calculate same padding. The classifier in the end additionally needs to know the number of labels in order to create the MLP.

Algorithm 4: Random CNN Creation

Require: $0 < D_{min} \leq D_{max}$ {Minimum and maximum CNN depth}
Require: I {Input shape}
Require: L_a {Set of available layers}
Require: N {Number of labels}
1: $D \leftarrow$ random_int(D_{min}, D_{max}) {CNN depth}
2: $L \leftarrow$ random_layer_stack(D) {Either of approach 1-3}
3: $O \leftarrow$ output_shape(L)
4: $C \leftarrow$new classifier(O, N) {Add SMMLP classifier}
5: **return** L, C

3.1.4 CNN Recombination

Part of the genetic algorithm combines the head and tail of two individuals using one-point cross-over, as depicted in figure 3.1. Both layer stacks of the feature detection part of the algorithm are split randomly. The head of one stack is joined with the tail of the other one and vice versa, creating two new individuals. A basic approach to do so is shown in algorithm 5.

This algorithm suffers from the same half pixel problem: The number of layers decreasing the resolution is limited, and not every recombination produces a valid individual.

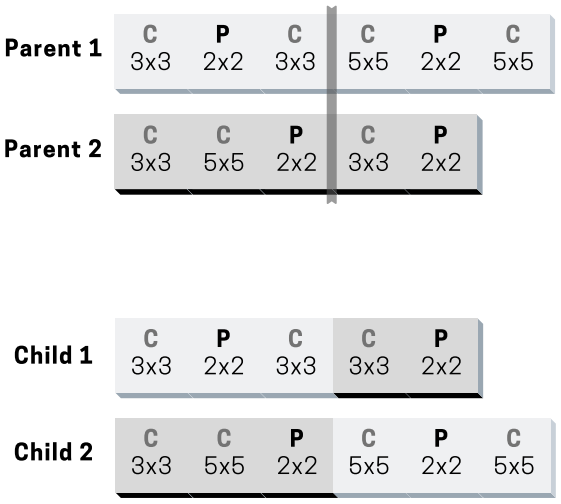


FIGURE 3.1: One-point crossover. The parental genomes, represented by the "C" and "P" genes (convolutional and pooling layers) and their kernel configuration, are sliced open at a random location and recombined cross-wise. This results in two new individuals.

Algorithm 5: Naive Recombination that crosses two layer stacks and creates a new one

Require: L_1, L_2 {Two layer stacks to be combined}
1: $H_1, T_1 \leftarrow random\_split(L_1)$
2: $H_2, T_2 \leftarrow random\_split(L_2)$
3: $C_1 \leftarrow \text{push}(H_1, T_2)$
4: $C_2 \leftarrow \text{push}(H_2, T_1)$
5: **return** C_1, C_2

The ideas of algorithm 1 and 2 can be reused for this problem, as shown in algorithms 6 and 7 respectively. They only implement joining the head and tail, the random split is performed in the same way as shown in algorithm 5.

Algorithm 6: Recombination parallel to algorithm 1

Require: H, T {Head and tail to be combined}
1: $C \leftarrow H$ {Child starts with head}
2: **for all** $t \epsilon T$ **do**
3: $C' \leftarrow \text{push}(C, t)$
4: **if** C' has invalid output shape **then**
5: **return** C
6: **end if**
7: $C \leftarrow C'$
8: **end for**
9: **return** C

The drawbacks of both algorithms are the same as algorithm 1 and 2. Either the resulting layer stack is shallow because of early termination, or the pooling layers have a higher

Algorithm 7: Recombination parallel to algorithm 2

Require: H, T {Head and tail to be combined}
1: $C' \leftarrow H$ {Child starts with head}
2: **for all** $t \epsilon T$ **do**
3: $C' \leftarrow \text{push}(C, t)$
4: **if** C' has valid output shape **then**
5: $C \leftarrow C'$
6: **end if**
7: **end for**
8: **return** C

density in the first part of the layer stack.

A third option is to modify algorithm 5 by recreating the head and tail stacks if the half pixel problem is detected, as shown in algorithm 8. Because it completely mitigates the drawbacks, is the most flexible, and does neither bias towards shallow or unevenly distributed pooling layers it has been used for the experimentation.

Algorithm 8: Recombination regenerates Head and Tail when Half Pixel Error occurs

Require: L_1, L_2 {Two layer stacks to be combined}
1: **repeat**
2: $H_1, T_1 \leftarrow random\_split(L_1)$
3: $H_2, T_2 \leftarrow random\_split(L_2)$
4: $C_1 \leftarrow \text{push}(H_1, T_2)$
5: $C_2 \leftarrow \text{push}(H_2, T_1)$
6: **until** C_1 and C_2 have valid output shape
7: **return** C_1, C_2

3.1.5 Half Pixels during Mutation

When adding, removing or manipulating a layer on the layer stack, the data in the pipeline may change shape. The output shape of the manipulated layer must be propagated consecutively to all following layers, the components may need to recalculate their padding, and the output shape of the last layer needs to be checked for half pixels.

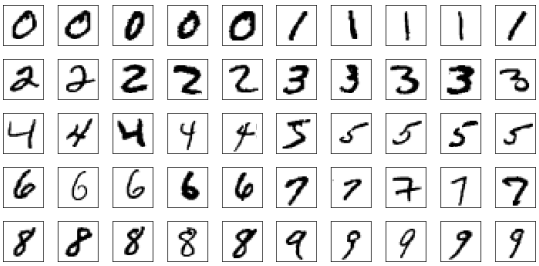
If half pixels are detected, the operation is aborted and will be randomised again.

3.2 Fitness Evaluation

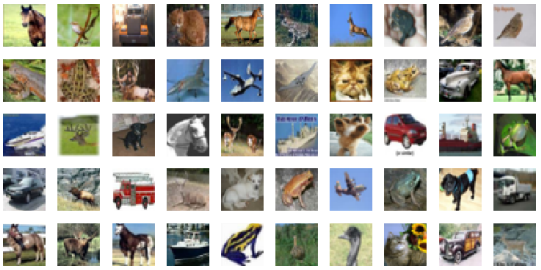
The genetic algorithm needs to evaluate individuals. Because individuals are topologies, a CNN needs to be created according to the topology genome, trained using machine learning on train data, and tested on the test data.

3.2.1 Data Sets

Ideally, the algorithm should work on any given data set. Two data sets of distinct difficulties were chosen to evaluate this thesis. Both data sets have been researched on extensively and allow comparisons to literature.



(A) MNIST[16] is a data set of handwritten digits. It has only one colour channel and is 28x28 pixels and represents easier problems.



(B) CIFAR10[13] is a data set of natural images of 10 classes. It has three colour channels, is 32x32 pixels, and represents complex problems.

FIGURE 3.2: Examples from the two data sets that the algorithm is trained and tested on

3.2.1.1 MNIST

The MNIST data set [16], examples shown in figure 3.2a, consists of 60,000 train and 10,000 test labelled handwritten characters between 0 and 9. Each character is 28 by 28 pixels and grey-scale.

It was picked because it is a rather easy task for a CNN - there are architectures with only six layers that achieve 99% accuracy after only minutes of training [12].

Following the best practises of [24, 3], the MNIST data set has been augmented with random rotations between -10 and 10 degrees (enlarging the image resolution), and a random crop to 32x32 pixels (with optional padding of white) ensures that the characters are of size 2^n (this is important for pooling layers, see section 2.1.4).

3.2.1.2 CIFAR10

CIFAR10 [13], examples shown in figure 3.2b, is a data set of 50,000 train and 10,000 test labelled natural rgb images of 10 classes, each 32x32 pixels. The classes are airplane, car, bird, cat, deer, dog, frog, horse, ship, and truck.

This data set is considered quite complex due to its variety of labels, complexity in the pictures, and noise. CNNs need to be hundreds of layers deep in order to perform above 90% accuracy ([9]).

Following [26, 24, 3], the data set is padded with 4 pixels and randomly cropped to 32x32 pixels, and a random horizontal flip is applied with a probability of 0.5.

3.2.2 Hyper Parameters

To increase speed by exploiting parallel computation, mini-batches (i.e. [12, 11]) were used. Because the networks operating on CIFAR10 are deeper and have three colour channels, they require more memory, therefore we configured them to use a smaller batch size of 50. The mini-batches for MNIST were configured to be of size 100.

[26] trains individuals for 350 epochs and decays the learning rate by 0.9 at the 1^{st}, 149^{th} and 249^{th} epoch. Because they did not use mini batches and because we needed to reduce wall time, we decreased the number of epochs. Our base experiment set the number of epochs to 60 and scaled down the learning rate decrement points accordingly to 1, 26, 43. For MNIST, we use 6 epochs and decrease the learning rate at epoch 1, 3 and 5.

Other hyper parameters for machine learning have been taken from [26]: Individuals are trained on GPUs using stochastic gradient descend minimising cross-entropy loss [21], with a learning rate of 0.1 and a momentum of 0.9.

3.2.3 Partial Training

As described in section 1.3, evolution by natural selection does not need an absolute fitness score, but rather a relative measure to separate the more fit from the less fit in each generation. We adapt the idea of partial training from [21] and train our models successively. The core idea is that earlier generations are trained for shorter durations than later, allowing us to spend less time on early unfit topologies. We hope to reduce training time and still maintain our result accuracy.

In contrast to [21], the weights are not inheritable, so partial training cannot be done implicitly. Instead, we introduce an epoch function. The epoch function defines how long unseen individuals are to be trained, and if already observed topologies need to be trained further.

The simplest type of epoch function is a linear function that only depends on time, i.e. the number of generations, going up between a lower and an upper bound. Such a function gives us control over the number of epochs and is easy enough to be tuned manually. It also allows us a direct migration from the flat number of epochs to a dynamic function. However, just as with a flat epoch number, we need to know the range of epochs a-priori. Also, it will introduce new hyper parameters for the two bounds that are dependent on the data set. More complicated functions are possible, for example one could react to previous fitness evaluations, but we will stick to the linear function because of its simplicity.

Experiment 4.6 will show that this method succeeds.

3.2.4 Fitness Cache

Training a CNN can take hours or even days. To prevent evaluating the same model over and over again, a fitness cache has been introduced.

In its simplest form (as described by [26]), the fitness cache is a map that matches a unique CNN description to a training score. For this description Sun et al. used hash values; our implementation uses the CNN's genome described in section 3.3.4.

This basic fitness cache was extended by storing both evaluations and trained CNN models. This is to support partial training. The CNN genome (defined in section 3.3.4) stores the number of training epochs, allowing the cache to save multiple stages of partially trained individuals per topology. If a topology has been previously evaluated on a smaller number of epochs, the partially trained model will be queried from the cache, cloned, trained further, and evaluated.

3.3 Evolution

3.3.1 GA Flow

Following [26], we implemented a generational GA of 20 individuals that is evolved over 20 generations.

An initial population of 20 random CNN topologies is created using algorithm 4. The objective fitness function assigns individuals their fitness score. This is done by creating, training and testing a CNN according to the genome; their test accuracy (i.e. the percentage of correctly classified unseen images) is assigned as their fitness score. If we were to use regularisation, the fitness score would be reduced according to a measure of complexity (see section 3.3.3).

Algorithm 9: Tournament selection

Require: P {Population of individuals}
1: **repeat**
2: $I_1 \leftarrow$ random_entry(P)
3: $I_2 \leftarrow$ random_entry(P)
4: **until** $I_1 \neq I_2$
5: **if** fitness(I_1) > fitness(I_2) **then**
6: **return** I_1
7: **else**
8: **return** I_2
9: **end if**

Two individuals are selected via tournament selection [17] (algorithm 9). We chose tournament selection because 1) its selective pressure is low, this will be shown vital in

our analysis, 2) it was successfully used in the past [26], and 3) it can cope with negative fitnesses without adaptation (in contrast to roulette selection), which is required for the regularisation method we use.

With probability P_c the two individuals are crossed over and with probability P_m they are mutated (details on these operations follow in section 3.3.2). If the best individual of the previous generation is not in the new one, it replaces the weakest individual of the new generation (elitism). The GA follows [26, 1] and is sketched out in algorithm 10.

Algorithm 10: Genetic Algorithm

Require: P_c {Probability of crossing over}
Require: P_m {Probability of mutating over}
Require: G {Number of generations}
Require: S {Population size}
$P \leftarrow$ random_population(S) {Initial population}
while $G > 0$ **do**
  $B \leftarrow$ best_individual(P)
  $P' \leftarrow []$ {New generation's population}
  while len(P') $< S$ **do**
    $P_1 \leftarrow$ tournament_selection(P)
    $P_2 \leftarrow$ tournament_selection(P)
    if rand() $\leq P_c$ **then**
      $P_1, P_2 \leftarrow$ crossover(P_1, P_2)
    end if
    if rand() $\leq P_m$ **then**
      $P_1 \leftarrow$ mutation(P_1)
    end if
    if rand() $\leq P_m$ **then**
      $P_2 \leftarrow$ mutation(P_2)
    end if
    $P' \leftarrow$ push(P', $[P_1, P_2]$)
  end while
  if $B \notin P'$ **then**
    $P' \leftarrow$ replace_weakest(P', B) {Elitism}
  end if
  $P \leftarrow P'$
  $G \leftarrow G - 1$
end while
return best_individual(P)

3.3.2 Genetic operators

Both genetic operators, cross-over and mutation, are used.

We set the probabilities for cross-over and mutation to $P_c = .9, P_m = .2$. This follows [26] and common GA conventions [4]. As shown in algorithm 10, cross-over and mutation

Table 3.2: Probability distribution for mutation

Operation	Probability
Insert skip layer	.7
Insert pool layer	.1
Remove layer	.1
Mutate layer	.1

are not mutually exclusive, meaning all permutations of the operators can occur: Neither (identical copy), just one, or both (with a fixed order of cross-over first, then mutation).

The mutation of a CNN is mutually exclusive, by which we mean we draw a random variable from a probability distribution and apply the matching function. The probability distribution is defined in table 3.2 and is taken from [26]. The "insert skip layer" mutation has a higher probability to motivate deepening CNNs.

Inserting a layer uses a random parameter configuration according to the search space: We configure random pool layers to have a 50/50 chance of either kernel function. As motivated in section 2.1, they always halve the input resolution by applying a 2x2 kernel with a 2x2 stride and valid padding. Skip layers were configured to not change the resolution by applying 3x3 filters on a 1x1 stride with same padding. Following [26], the number of filters is a uniformly random choice from the set $\{64, 128, 256\}$.

Pooling layers mutate by swapping their kernel function. Skip layers mutate by re-randomising their kernel (which can result in the same configuration).

For cross-over, one-point cross-over operates on the layer stack of the feature recognition part (not the SMMLP and softmax), as motivated in section 2.2.1 and described in section 3.1.4. The SMMLP and softmax are always appended once in the end.

3.3.3 Regularisation

One of the main concerns with the algorithm is the computational overload, mainly the time required to train individuals. To reduce this, experiment 4.5 will show that regularisation within the GA helps.

We want to primarily decrease the GPU time the algorithm spends on training and testing. During fitness evaluation, we therefore measure the time required from the beginning of training to the end of testing. For every hour spent, we decrease the fitness linearly by 0.05. This means that if two individuals achieve the same accuracy, but one spends an hour longer than the other, the fitness of the slower one will be decreased by 0.05.

This method of regularisation was chosen because it feels quite intuitive and it tackles the desired effect directly. If we used some other measurement, for example the number of parameters, tuning becomes non-trivial.

The drawback of using GPU time as a regularisation is that this measurement is relative to the hardware used - a faster GPU will result in less punishment of the same individual. It is therefore important that the GPUs per experiment are similar and have no drastic difference in workload from other applications. Further, it means that the results between different system configurations will differ.

Another potential drawback of using this regularisation is that individuals might be assigned a negative fitness score, which would complicate roulette selection. Throughout this work, we use tournament selection that is compatible with negative scores without adaptation.

Lastly, introducing yet another hyper parameter makes the algorithm even more complicated. We argue that the parameter is worth it because results of experiment 4.5 will show a performance gain while maintaining accuracy.

3.3.4 Genome representation

As motivated in 1.3, a bitwise representation of a CNN like in [29] complicates both mutation and human understanding. We therefore introduce a genomic representation that is flexible enough for future applications, can be easily used for cross-over, and is quickly understandable.

Following the structure of 2.1, we will describe CNN components hierarchically and arrange them bottom-up. The top-most component is the CNN, its genome definition will be used to represent individuals in our GA, as an identifier for the fitness cache, and to document findings of our experiments. Each component has a separate search space that defines what parameters random creation and mutation can change, as described in chapter 4. The search space was decoupled from the rules of our genomic representation to make the genome syntax reusable for future applications.

We specify the genome for each component with a context-free semantic specified in *Backus-Naur form* (BNF) [19]. This form specifies *rules*, indicated by ::=, that define to substitutions into other rules or *terminals* are to be applied. Rules are delimited by <> and terminals by ″. If more than one substitution is allowed, they are delimited by a | (either in the same line or following a line break). Comments are separated by a ; character. All rules must be substituted into terminals; an expression is implementing a grammar correctly if there is at least one way to substitute all rules into terminals, and all terminals joined to a string is equal to the expression in question.

As a convention, we denote rules that are reused for other components in uppercase. The first reused rule is the definition of natural numbers, formally defined as:

⟨*NN*⟩ ::= ⟨*leading-digit*⟩
| ⟨*leading-digit*⟩ ⟨*digit*⟩

⟨*leading-digit*⟩ ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

⟨*digit*⟩ ::= '0' | ⟨*leading-digit*⟩
| ⟨*digit*⟩ ⟨*digit*⟩ ; recursively allow long numbers

Kernels

We define the genome syntax of a kernel (or filter) by the following BNF:

⟨*KERNEL*⟩ ::= ⟨*kernel-size*⟩ ' ' ⟨*stride*⟩ ' ' ⟨*padding*⟩

⟨*kernel-size*⟩ ::= ⟨*NN*⟩ 'x' ⟨*NN*⟩ ; the kernel size vector delimited by an x

⟨*stride*⟩ ::= ⟨*NN*⟩ 'x' ⟨*NN*⟩ ; the stride vector delimited by an x

⟨*padding*⟩ ::= 'S'|'V' ; same or valid padding

A kernel of size 3x3 with a unit stride and same padding would therefore be represented as '3x3 1x1 S'. We did not incorporate the exact padding number in pixels into the genome because same padding depends on the input size of the layer; instead the pixel value will be calculated dynamically using (2.2).

Convolutional Layer

The convolutional layer is represented by:

⟨*CONV*⟩ ::= 'C[' ⟨*filters*⟩ '(' ⟨*KERNEL*⟩ ')]'

⟨*filters*⟩ ::= ⟨*NN*⟩

A convolutional layer with 256 filters of the same definitions as the previous example is described as 'C[256(3x3 1x1 S)]'. Note that the activation function ReLU and batch normalisation are implicit and not part of the genome.

Skip Layer

The skip layer joins at least two convolutional layers:

⟨*SKIP*⟩ ::= 'S{' ⟨*layer*⟩ ';' ⟨*layer*⟩ '}' ; at least two layers
⟨*layer*⟩ ::= ⟨*CONV*⟩
| ⟨*layer*⟩ ';' ⟨*layer*⟩ ; or more than two

A skip layer with two convolutional layers, the first being configured with the same definitions as the previous example and the second one with 64 feature maps is therefore represented by 'S{C[256(3x3 1x1 S)];C[64(3x3 1x1 S)]}'. Adapters are not part of the genome as they depend on the input shape, they are added by the algorithm implicitly.

Pooling Layer

We define the genome of a pooling layer with the following BNF:

⟨*POOL*⟩ ::= 'P ' ⟨*pooling-function*⟩ '[' ⟨*KERNEL*⟩ ']'
⟨*pooling-function*⟩ ::= 'AVG' | 'MAX'

An average pool layer with a 2x2 kernel and stride is described with 'P AVG[2x2 2x2 V]'. Note that pool layers normally have valid padding as they are supposed to reduce the resolution, however this is not enforced by the grammar.

Softmax MLP Classifier

Because the number of neurons in the input and output layers of the MLP are defined implicitly through the preceding layers and the number of output labels, it is not part of the genome. The flatten process is not part of the genome either as it is syntactically required for the input layer of the MLP. The genome only describes if hidden layers are used and if so how many neurons they include:

⟨*SMMLP*⟩ ::= 'SMMLP[' ⟨*layers*⟩ ']'
⟨*layers*⟩ ::= '' ; no hidden layers
| ⟨*hidden-layer*⟩ ; one or more hidden layers
⟨*hidden-layer*⟩ ::= ⟨*NN*⟩ ; the number of neurons in this layer
| ⟨*hidden-layer*⟩,⟨*hidden-layer*⟩

A softmax MLP classifier without any hidden layers is therefore described as 'SMMLP[]'. If it had two hidden layers of 100 and 50 neurons each, it would be described as 'SMMLP[100,50]'.

CNN

Lastly, the CNN joins all the components described above in a sequential layer architecture. For the feature detection part it joins the three different layers to a list, and for the classification part it adds a SMMLP. The number of training epochs has been incorporated as well in order to allow evolving training epochs. Other parameters might be added as needed.

We denote carriage returns that are part of the grammar by '\n'

$$
\begin{aligned}
\langle cnn \rangle &::= \langle detection \rangle \text{ '\textbackslash n'} \\
&\quad \langle classification \rangle \text{'\textbackslash n'} \\
&\quad \langle epochs \rangle \\
\langle detection \rangle &::= \langle detection \rangle \text{ '\textbackslash n'} \\
&\quad \langle detection \rangle \text{ ; arbitrary number of layers} \\
&\quad | \ \langle CONV \rangle \\
&\quad | \ \langle SKIP \rangle \\
&\quad | \ \langle POOL \rangle \\
\langle classification \rangle &::= \langle SMMLP \rangle \\
\langle epochs \rangle &::= \langle NN \rangle
\end{aligned}
$$

As an example, this is a very simple CNN topology:

```
S{C[256(3x3 1x1 S)];C[256(3x3 1x1 S)]}
P MAX[2x2 2x2 V]
SMMLP[]
60
```

Listing 3.1: A sample CNN genome

This genome representation allows to visualise mutation and cross-over easily. Cross-over can just slice open the detection layer stack and recombine. The effects of mutation can be fine-tuned and balanced. The grammar can be easily extended to involve new layer types and parameters. Because of the recursive definitions of BNF rules, the grammar can be integrated into software architecture and allow to create and manipulate CNNs hierarchically just using string manipulation and search space definitions.

3.3.5 GA Visualisation

Understanding what happens in a GA often a non-trivial task. A common way of visualisation is to plot out time (wall time, generation) versus performance (fitness, accuracy), i.e. [21, 29]. This visualisation however does not show what the GA is doing, it only shows the results obtained. To verify if and how the GA works and how well

it performs, especially to explore modifications to the algorithms, a (to our knowledge) new visualisation was developed by extending a standard generation/fitness graph to include ancestral trees.

Every time we cross over two individuals, the child gets a reference to each of its parents; children from asexual reproduction get a reference to one parent. We also save information to the exact mutation, i.e. which layer type was inserted. In the generation/fitness graph, each child is connected to their respective parent(s). This visualises which individuals were selected for reproduction and which operation was successful or unsuccessful.

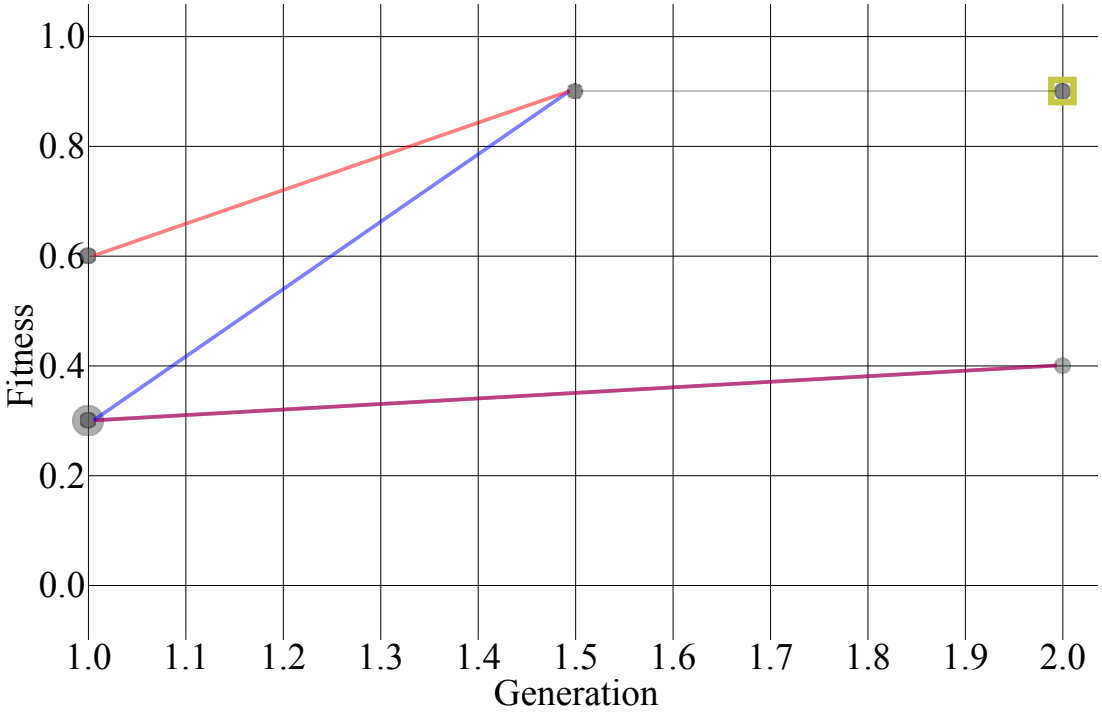


FIGURE 3.3: Example Visualisation of artificial data. The golden square indicates the best individual, thick red and blue lines cross-over, thin grey lines asexual reproduction. Thick purple lines are both red and blue lines laying on top of each other. If both cross-over and mutation occurs, the intermediate individual is shown at a pseudo generation in between with the fitness of its child.

The example graph, figure 3.3, shows six distinct genomes (CNN topologies) that were evolved over two generations, the first generation being an initial random population.

The best individual (more precisely, the first occurrence of the maximum fitness value) is marked by a golden square. The best individual therefore scored a fitness of 0.9 in generation 2.

The best individual is connected to one parent with a fine grey line, indicating it was asexually produced (mutation or elitist clone). Its parent is in the *pseudo generation* 1.5, which means that it was never evaluated by the objective fitness function and is an intermediate individual. This happens when sequentially applying both genetic

operations (cross-over and mutation). Because this parent was never evaluated, it is impossible to draw it on the Y axis. To still visualise it, its fitness is (most likely wrongly) assumed to be the fitness of its child.

The parents of this individual are connected with a thicker red and blue line, indicating a cross-over. The two parents belong to generation 1, indicating they are part of the initial random population.

The parent connected by a blue line (the one with lower fitness) has another individual with a different genome at the same location, represented by a second, bigger circle. The size of the circle is no qualitative measure, it is only used to visually indicate that multiple distinct genomes are at that specific coordinate.

The last individual, the one at generation 2 that is not marked by the golden square, is connected to its parents with what appears to be a thicker purple line. This line is just a thicker blue and thicker red line laying on top of each other, indicating that its parent topology was crossed over with itself.

This representation gives a lot of insight. However, some details are still missing, for example the genomes, what the mutation did to them, accuracy (if accuracy is not equal to fitness), and measures of complexity such as time, size, or number of parameters. Also it fills up quickly and might get too cluttered, especially for bigger population. To investigate these features and the ancestry tree of specific individuals, an interactive, zoomable graph [1] has been developed where individuals can be selected to highlight their genome, exact mutation, metrics, and ancestry tree.

3.4 Technologies and Hardware used

For GPU accelerated CNNs, the python library pytorch [20] was used because it implements core CNN functions on GPUs while giving a lot of control over the hyper parameters. The GA was custom implemented in python to allow maximum control.

The experiments were run on the Iridis 5 HPC, either on 4 GTX1080 Consumer GPUs or 2 Volta V100 Enterprise GPUs (as documented in the experiments). Each GPU was used to concurrently evaluate one individual at a time.

The interactive visualisation was implemented using standard web technologies and the libraries D3 for graph visualisation and jsdiff to visualise genomic differences.

[1] The interactive visualisations of the experiment can be accessed at ga.yaron-strauch.com

Chapter 4

Experiments

All experiments extend the base configuration shown in table 4.1. All parameters not described in the following experiments use this base configuration as a default. Results of the experiments will be evaluated by comparing them to the results from the base configuration; the base configuration will be compared to relevant literature.

GAs are naturally very stochastic. In order for us to conclude that an approach definitely does or does not work, we would need to extensively repeat experiments and compare mean values. Because of the extensive wall times of the algorithm, this is not possible, and ways to reduce wall time need to be developed. In order to develop measures to reduce wall times, experimentation is needed. This egg and hen problem cannot be solved, and it was decided to do a one-shot approach where one run is used to get a general direction in what could work or what could not work. This however means that all results presented in this work have no statistical significance and results should be taken as an indication for future research.

Following common practices, we report the accuracy of a CNN as a percentage (0%-100%) of correctly classified images from the unseen test set. To compare accuracies of two CNNs, we do not use relative (percentage-wise) comparisons because they would lose their absolute scale. Instead we subtract the accuracies from each other. To avoid confusion, we report this difference as a fraction. For example if two CNNs have accuracies of 50% and 75% respectively, we say that the second one has an accuracy that is 0.25 higher (not 50% better).

4.1 Base Configuration

Table 4.1: Base Configuration

Parameter	Configuration	
Data Set	CIFAR10	
Population size	20	
Number of generations	20	
CNN depth	Uniform random number between $[10, 120]$	
CNN feature det. components	{Pooling layer, skip layer}	
CNN layer initialisation	Algorithm 1 (stop adding layers on half pixels)	
Genetic operators	Crossover	.9
	Mutation	.2
CNN mutation	Insert skip layer	.7
	Insert pool layer	.1
	Remove layer	.1
	Mutate layer	.1
Skip layer search space	Number of convolutional layers	2
	Number of filters	Random choice of {64, 128, 256}
	Filter configuration	'3x3 1x1 S'
	Mutation	Re-randomise filter
Pooling layer search space	Kernel function	Random choice of {AVG, MAX}
	Kernel configuration	'2x2 2x2 V'
	Mutation	Swap kernel function
MLP	'[]'	
Training epochs	60	
Batch size	50	
Learning rate	.1	
Learning rate decay points	{1, 26, 43}	
Learning rate decay factor	.9	
Momentum	.9	
Regularisation	None	
GPUs	4 GTX1080 Consumer	

The first experiment, our base experiment, is similar to [26], except for some parameters that were changed. The implementation presented here uses a CNN depth between 10 and 120 layers, Sun et al. did not document an upper bound, however for computational reasons such a bound was necessary. Also, as described in section 3.2.2, we introduced mini batches, cut the number of training epochs down, and adjusted the learning rate decay points accordingly. Lastly, as described in section 2.1.5, the classifier was changed to also include a SMMLP.

4.1.1 Results

The GA ran for 15 GPU days and is visualised in figure 4.1.

The best individual found, in generation 17, has an accuracy of 88.36%, is 10 layers deep, and has 2.67 million parameters. Its genome is shown in listing 4.1.

```
S{C[256(3x3 1x1 S)];C[256(3x3 1x1 S)]}
S{C[128(3x3 1x1 S)];C[256(3x3 1x1 S)]}
P AVG[2x2 2x2 V]
S{C[256(3x3 1x1 S)];C[256(3x3 1x1 S)]}
S{C[064(3x3 1x1 S)];C[256(3x3 1x1 S)]}
P AVG[2x2 2x2 V]
P AVG[2x2 2x2 V]
P AVG[2x2 2x2 V]
P AVG[2x2 2x2 V]
SMMLP[]
60
```

LISTING 4.1: The best topology found uses more skip layers in the beginning and more pool layers in the end. Skip layers are only applied pair-wise. The network is not very deep.

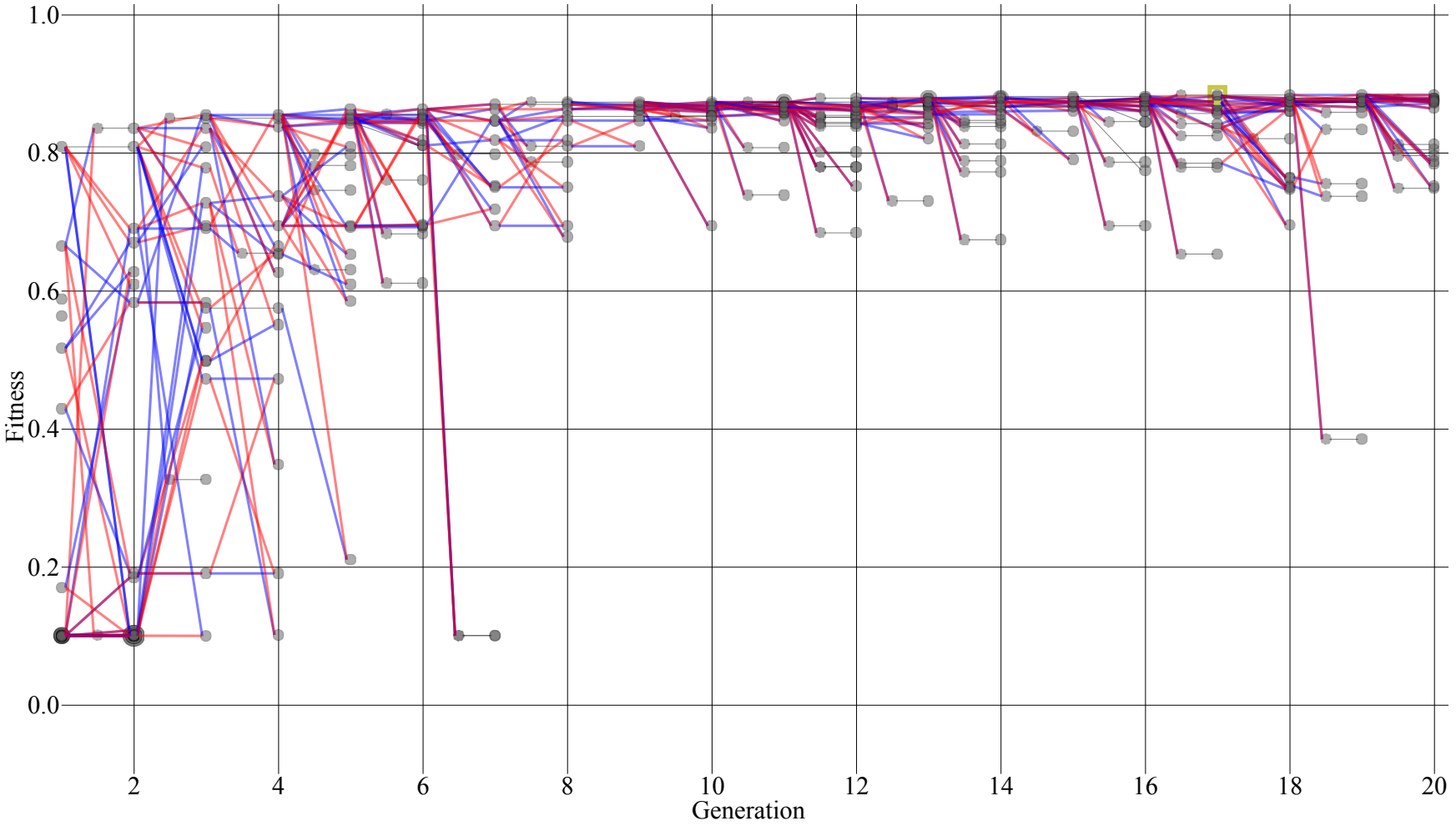


FIGURE 4.1: The base experiment has a good variety of individuals in all generations. The worst fitness is 0.1 (corresponds to random chance), selective pressure eliminated most inferior topologies under 0.6 within 5 generations without collapsing the variance completely. The best individual is found after 17 generations. After 8 generations, the increase in fitness stagnates and new better individuals are found less frequently.

From the GA tree (figure 4.1), we can see that the fitness of the initial population spreads around 0.1-0.8, with an inferior cluster in the first two generations at a fitness of 0.1. In a classification of 10 labels, this corresponds to random chance. This cluster is thinned out and eliminated quickly - in generation 3 and later, none of the individuals of said accuracy were selected for reproduction.

In generation 2, the best individual with a fitness of 0.84 is a product of a cross-over of the second best individual of the previous generation and an individual from the inferior cluster, showing that the desired jumps in fitness due to cross-over are happening. This specific instance notably used an individual with a fitness of random chance to achieve a jump in fitness. Most of the cross-over maintains a variety in fitness scores.

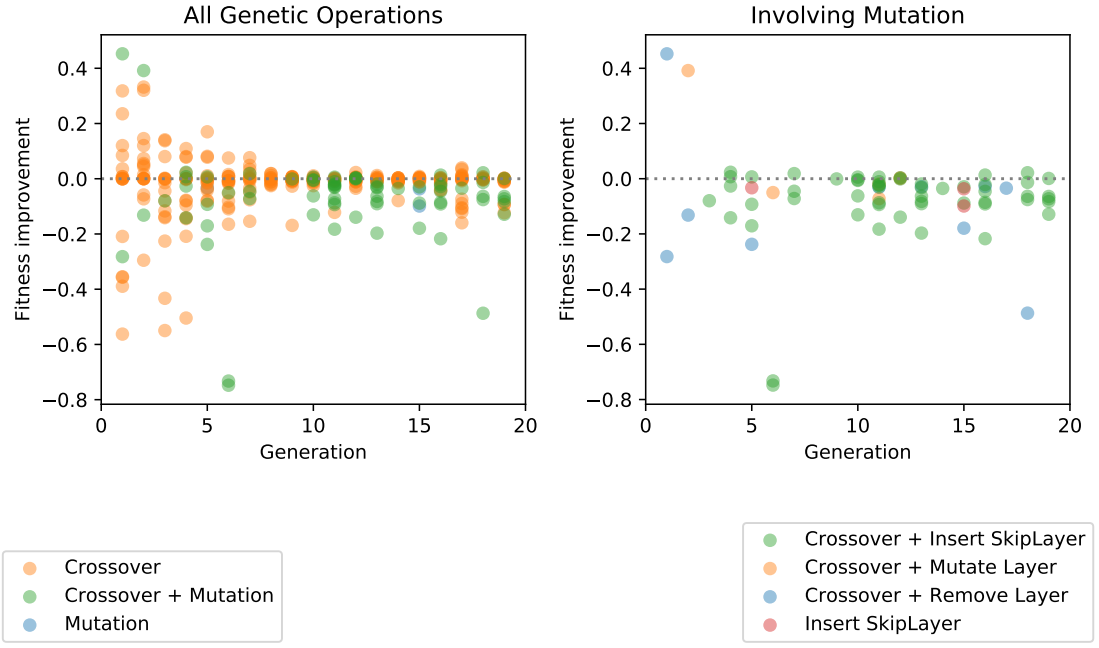


FIGURE 4.2: The fitness improvement per individual shows that early mutations are much more successful than later on. Most of the genetic operators in generation 9 onwards have a neutral or negative fitness impact. The biggest jumps in fitness involve cross-over. Almost no asexual offspring was generated, and of those that have (the red insert SkipLayer operations), no mutation was successful.

Figure 4.2 plots all genetic operations and their relative fitness improvement in relation to the mean fitness of their parent (asexual reproduction), parents (cross-over), or grandparents (crossover followed by mutation). We can see that most genetic operations involve cross-over, which is as expected because we set the cross-over chance to 0.9. There is almost no asexual offspring generated due to the big cross-over chance, so not much can be said about pure mutation except for that all of the pure mutations inserted a skip layer, and every time that mutation was unsuccessful.

The first graph in figure 4.3 illustrates the convergence of the population. The max fitness is only increasing (elitism), and the mean fitness is generally increasing with

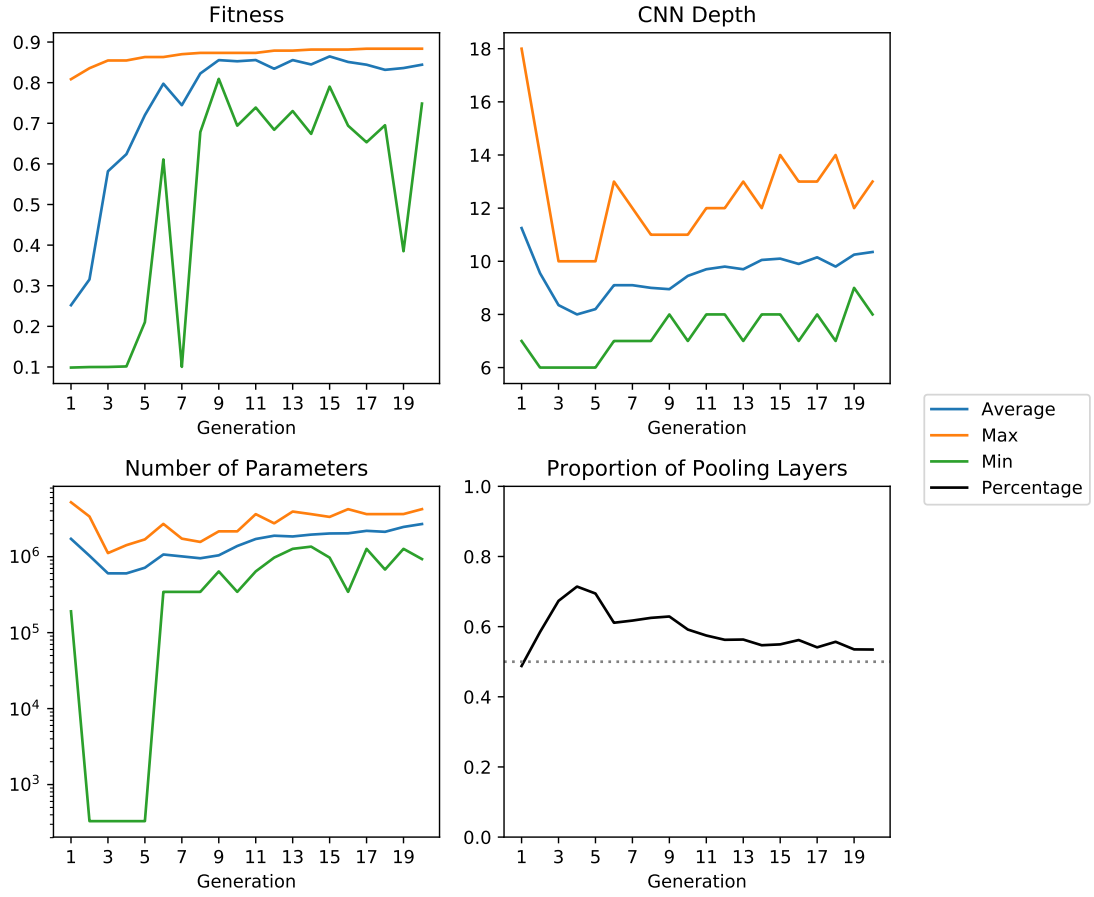


FIGURE 4.3: The mean and max fitness converge well. The CNN depth, number of parameters and proportion of pooling layers have a rough parable shape, the proportion of pooling layers is anti proportional to the two measures of complexity. The CNN depth plummets for the first 5 generations until it increases again. In generation 20, most CNNs have 10 layers. The proportion of pooling layers starts at 50%, jumps up and converges to around 55%.

some minor step backs. Both curves converge well. The min fitness jumps back and forth, illustrating that some mutations are very disadvantageous.

The proportion of pooling layers is anti proportional to the two measures of complexity (CNN depth and parameter count), which makes sense as pooling layers have less parameters then skip layers. The proportion starts at a roughly 50%, which is due to our initialisation logic (algorithm 1). It seems to be advantageous to have more pooling than skip layers. The CNN depth starts at a mean of less than 12, which is considerably smaller than the configured depth of 10-120 (mean of 65). This is due to the drawback of the initialisation algorithm that stops early if it detects half pixels (algorithm 1).

During the first 5 generations, the depth and number of parameters decrease quite drastically, until they start converging up again like a parabola. The later rise in complexity to a mean depth of 10 shows that our configured number of epochs can train networks

of this depth, but why did it fall to a mean depth of 8 before? After thorough investigation, no hidden variable that could cause such a non-linearity was found. A possible explanation is that deeper networks have a bigger search space of possible configurations (i.e. for the given search space, each skip layer brings two convolutional layers, each with three possible filter depths). Deeper networks are more likely to be inferior when created randomly, explaining the big drop of complexity in the beginning when selection removes them. Instead of using random chance to produce deep networks, the algorithm gradually deepened networks that were shallower but higher in fitness. Successive deepening was one of the ideas of [26], which is why the mutation probability distribution favours adding skip layers. Whether this theory indeed is the explanation for the non-linear convergence of complexity cannot be demonstrated without additional expensive grid search.

The algorithm strongly favours the "average" kernel function and evolved to use bigger skip layer filters, as figure 4.4 illustrates.

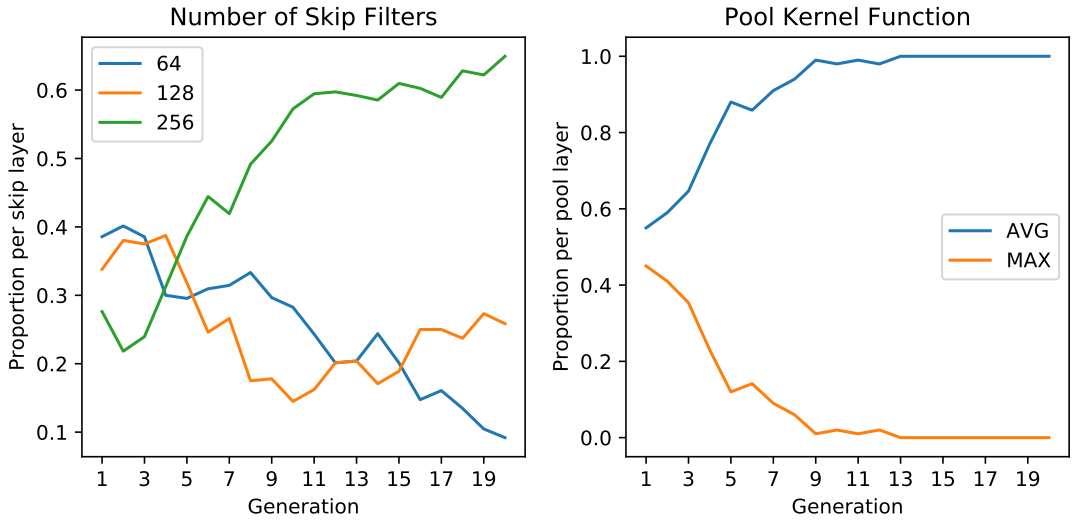


FIGURE 4.4: The filters within the skip layers evolved from an equal distribution in favour of big filters. The kernel function converges towards the "average" function monotonically.

The convergence of the kernels towards the "average" function is surprising, especially because hand-crafted networks such as ImageNet or ResNet use the max pooling function on CIFAR [14, 9]. Clearly, for the reduced search space of our algorithm, the average function is superior. The number of filters evolved in favour of 256 and against the smallest number of 64. About 25% of skip layers use 128 filters.

The distribution of pooling and skip layers has been further investigated. Figure 4.5 shows distributions of the two layer types in respect to depth.

We can see that the population clearly converges to preferring skip layers in the first and pool layers in the second half. Also, in generations 10 and 20 there is not a single

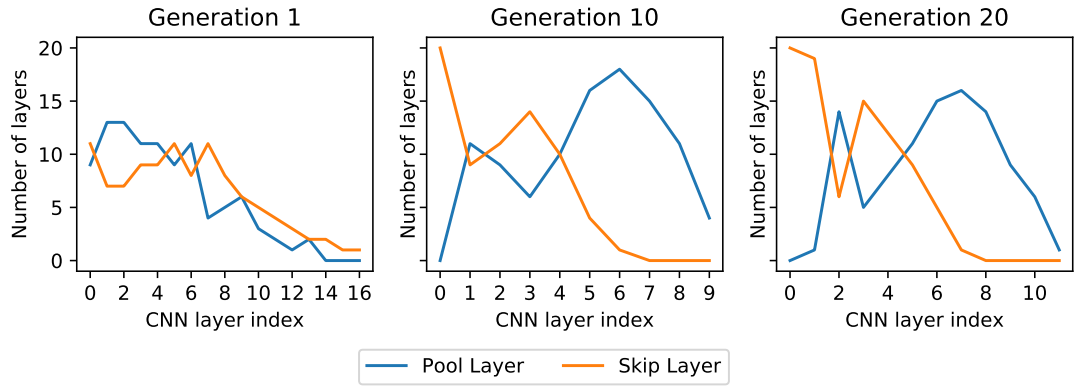


FIGURE 4.5: The population converges to have more skip layers in the first and more pooling layers in the second half of the genome. In generation 10 and 20, all individuals start with a skip layer. In generation 20, there is pattern of alternating skip and pooling layers at depth 1-6.

individual with a pool layer as its first layer. This makes sense because an early pool layer would reduce details of the images early and lose information.

A remarkable effect is that the population learned to alternate skip and pool layers. This can be seen in generation 20 of figure 4.5 - the distribution curves meet three times between the first and fifth layer index. Alternating pool and conv/skip layers is a pattern found in hand-crafted networks (i.e. [16]), however the best topology found alternates them pair-wise (see i.e. listing 4.1). This is not a common design pattern used by humans.

4.1.2 Comparison to literature

Table 4.2 compares our results to the two closest related CNN topology algorithms.

As mentioned earlier, this experiment is similar to Sun. et al [26]. Their algorithm ran for more than twice the time, and their result is 0.07 higher. As the number of parameters in their best model is comparable to ours, it is suspected that the discrepancy is due to the number of epochs. Our algorithm learned for 60 epochs on mini-batches of 50, their algorithm trained for 350 epochs without mini-batches.

Because there is no "conversion rate" between mini batch sizes, and due to the high stochasticity of the algorithm, we cannot estimate how much higher the results would have been given longer training time.

Xie and Yuille trained for 50 generations and 17 GPU days and achieved a result 0.11 worse than ours [29], despite having trained for longer. This might be due to a variety of factors since their implementation differs from ours. Most prominently, they use a binary representation and mutation, rendering mutations and cross-over to be less

TABLE 4.2: Comparing the base experiment to literature

Origin	Parameter / Measurement	Value
Sun et al. [26]	Accuracy	95.22%
	Number of generations	20
	Training epochs	350
	Batch size	1
	GPU Days	35
Ours	Accuracy	88.36%
	Number of generations	20
	Training epochs	60
	Batch size	50
	GPU Days	15
Xie and Yuille [29]	Accuracy	77.19%
	Number of generations	50
	Training epochs	180
	Batch size	1
	GPU Days	17

predictable, and their selection algorithm eliminates weak topologies and applies fitness-proportionate selection, increasing selective pressure substantially.

4.2 CNN Initialisation Approach 2

This experiment uses approach 2 for random CNN initialisation. Recapping section 3.1.2.2, the CNN layer stack gets initialised by not completely stopping the initialisation process when half pixels are detected, but only appending layers that do not produce half pixels. For our search space, this means that in the first part of the genome, we expect a 50/50 ratio of pooling to skip layers, and on deeper layers we expect skip layers only. We expect much deeper networks than the ones found in the base experiment, hopefully resulting in higher accuracies.

4.2.1 Results

The GA ran for almost 38 GPU days and is visualised in figure 4.6.

The best individual found, in generation 13, has an accuracy of 12.94%, is 51 layers deep, and has 19.32 million parameters. Its genome is shown in listing 4.2.

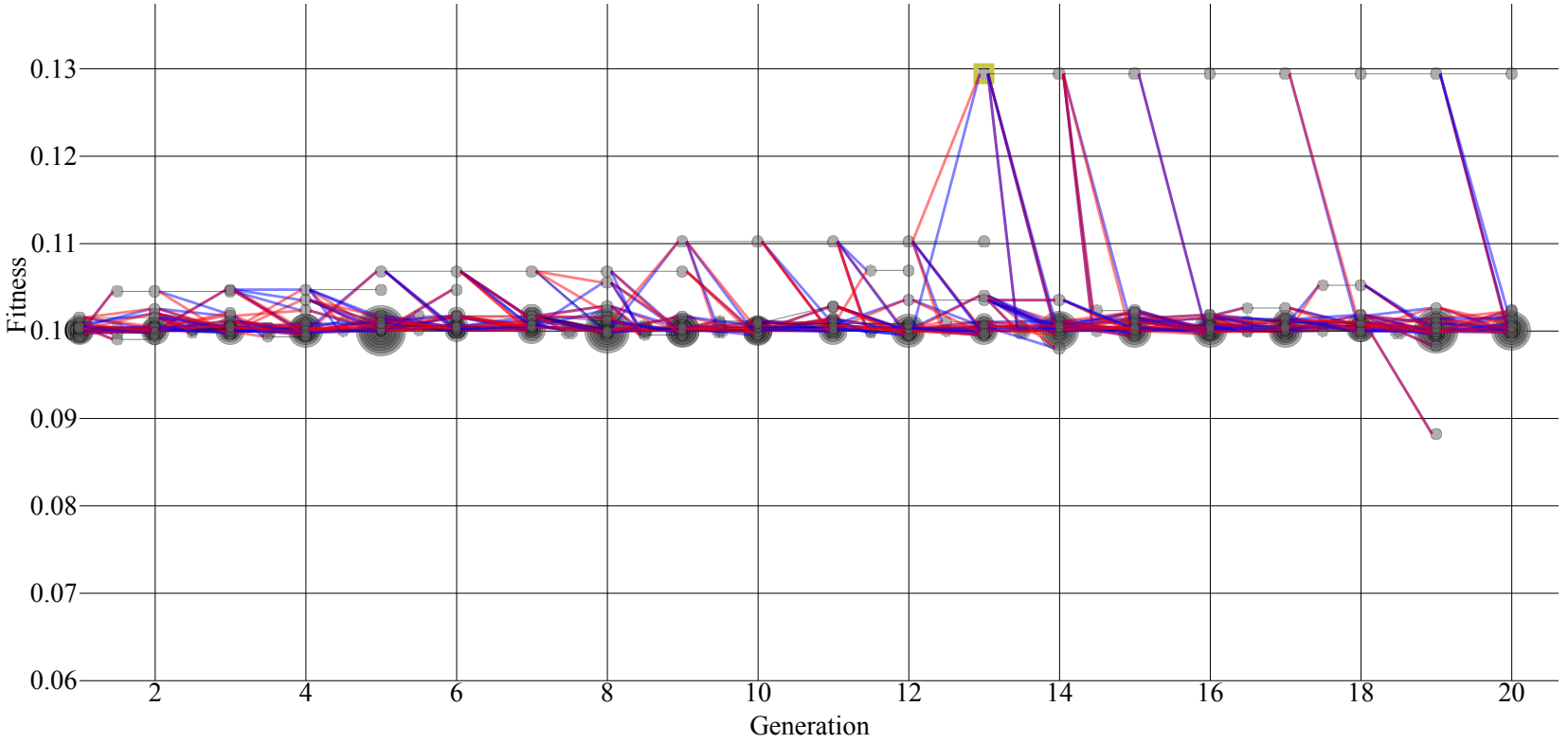


FIGURE 4.6: There are a lot of topologies that evaluate with an accuracy of 10% (random chance), even at the last generation. There is almost no considerable improvement and all topologies evaluate very poorly. When the best individual is found in generation 11, it is used for multiple cross-over approaches, all without success. In generation 19, there is even an individual with worse performance than random chance.

```
S{C[256(3x3 1x1 S)];C[256(3x3 1x1 S)]}
P MAX[2x2 2x2 V]
P MAX[2x2 2x2 V]
S{C[256(3x3 1x1 S)];C[256(3x3 1x1 S)]}
S{C[256(3x3 1x1 S)];C[256(3x3 1x1 S)]}
S{C[128(3x3 1x1 S)];C[128(3x3 1x1 S)]}
P MAX[2x2 2x2 V]
S{C[064(3x3 1x1 S)];C[256(3x3 1x1 S)]}
S{C[064(3x3 1x1 S)];C[256(3x3 1x1 S)]}
S{C[128(3x3 1x1 S)];C[256(3x3 1x1 S)]}
P AVG[2x2 2x2 V]
S{C[128(3x3 1x1 S)];C[256(3x3 1x1 S)]}
P AVG[2x2 2x2 V]
S{C[128(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[064(3x3 1x1 S)];C[064(3x3 1x1 S)]}
S{C[064(3x3 1x1 S)];C[256(3x3 1x1 S)]}
S{C[064(3x3 1x1 S)];C[256(3x3 1x1 S)]}
S{C[064(3x3 1x1 S)];C[256(3x3 1x1 S)]}
S{C[256(3x3 1x1 S)];C[064(3x3 1x1 S)]}
S{C[064(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[128(3x3 1x1 S)];C[256(3x3 1x1 S)]}
S{C[256(3x3 1x1 S)];C[256(3x3 1x1 S)]}
S{C[064(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[064(3x3 1x1 S)];C[256(3x3 1x1 S)]}
S{C[128(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[128(3x3 1x1 S)];C[064(3x3 1x1 S)]}
S{C[064(3x3 1x1 S)];C[064(3x3 1x1 S)]}
S{C[256(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[128(3x3 1x1 S)];C[064(3x3 1x1 S)]}
S{C[128(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[128(3x3 1x1 S)];C[064(3x3 1x1 S)]}
S{C[064(3x3 1x1 S)];C[064(3x3 1x1 S)]}
S{C[256(3x3 1x1 S)];C[256(3x3 1x1 S)]}
S{C[256(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[256(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[256(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[064(3x3 1x1 S)];C[064(3x3 1x1 S)]}
S{C[128(3x3 1x1 S)];C[064(3x3 1x1 S)]}
S{C[128(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[128(3x3 1x1 S)];C[064(3x3 1x1 S)]}
S{C[064(3x3 1x1 S)];C[064(3x3 1x1 S)]}
S{C[256(3x3 1x1 S)];C[256(3x3 1x1 S)]}
S{C[256(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[128(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[256(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[128(3x3 1x1 S)];C[256(3x3 1x1 S)]}
S{C[128(3x3 1x1 S)];C[064(3x3 1x1 S)]}
S{C[064(3x3 1x1 S)];C[064(3x3 1x1 S)]}
S{C[064(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[256(3x3 1x1 S)];C[064(3x3 1x1 S)]}
SMMLP[]
60
```

LISTING 4.2: With 51 layers, the genome is deep. We can clearly see that pooling layers are poorly spaced out, (almost) the last two thirds of the genome are skip layers.

From the GA tree (figure 4.6) and the fitness curve from figure 4.7, it becomes imminent that the algorithm does not yield any usable results. The jump in fitness in generation 13 is only increasing the population fitness once, but the mean accuracy does not converge up. The children of the best individual do not improve. This all indicates that there was random noise in the objective fitness function, for example the initialisation logic of the weights randomly produced something lucky. Weights are not inherited, and children

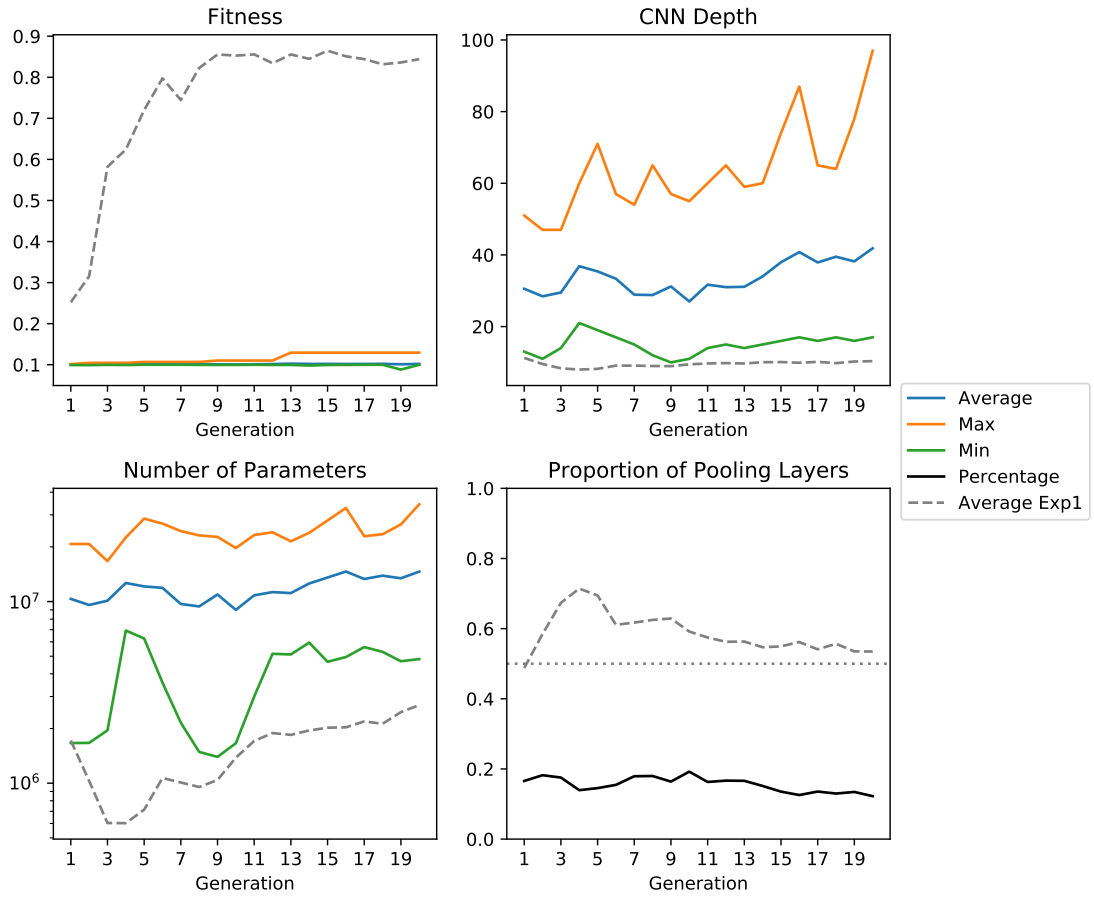


FIGURE 4.7: The average fitness stays around random chance - even when the max fitness jumps in generation 13, the mean fitness does not converge. The CNN depth and number of parameters increases over time, into the opposite direction than our results from the search space. The proportion of pooling layers is very low and decreases a bit.

cannot profit from this random side effect. In generation 19 there is even one individual with considerably less accuracy than random chance.

The fitness and CNN complexity fluctuates around the same value for the first 13 epochs; there are some short break-outs but there is no trend. In generation 13, the best individual is found, and with a depth of 51 (listing 4.2) it is substantially deeper than the average population of 35. Selective pressure causes the population to converge to become deeper instead of shallower; this does not work as we can see from the fitness curve. The increasing depth explains the declining pooling ratio, as all networks are working on the maximum number of pooling layers.

Arguably, the results might increase with the number of training epochs. Training the CNNs for longer might mitigate issues rooted in the depth of CNNs, but as this whole process already ran for almost 38 GPU days, increasing epochs is hardly justifiable.

The genome of the best individual, listing 4.2, clarifies the problem of unevenly distributed layers: Almost the last two thirds of the layer stack consist of skip layers only.

This problem is not specific to this individual, as figure 4.8 shows. In generation 20 there is no individual that has pooling layers in depth 15 or deeper.

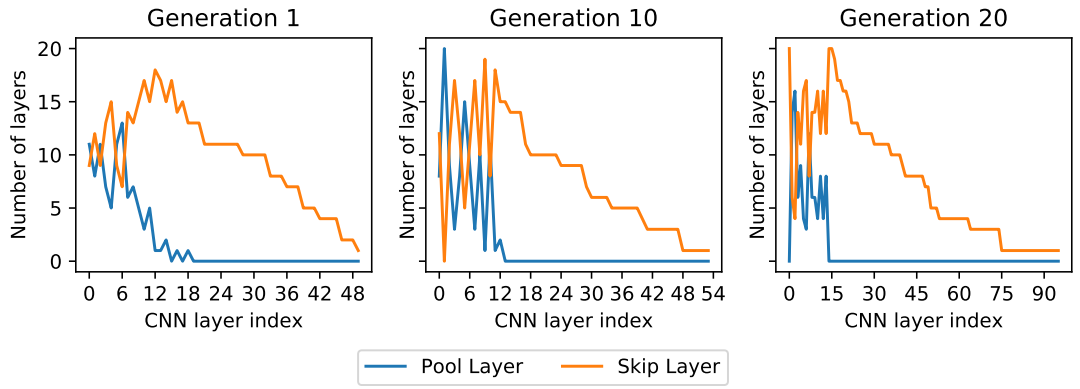


FIGURE 4.8: The pool layer count falls to zero on deeper layers and cannot be recovered during the evolutionary process.

Cross-over cannot fix the uneven distribution of pooling layers: Because it always combines a first part with a second part of the genome, the second part will always have no or next to no pool layers. Only mutation could fix that, but the speed of mutation is definitely too slow.

We consider this approach to be unfit and do not use it any further.

4.3 CNN Initialisation Approach 3

Following section 3.1.2.3, algorithm 3 is evaluated. This approach changes the probability distribution for adding pooling or skip layers, so that the networks are deep and mitigate the issues demonstrated in experiment 4.2. If half pixels would occur, we stop appending CNN layers completely. We expect to have deep layer stacks with evenly distributed pooling layers that hopefully have much better accuracies than the previous approach.

4.3.1 Results

The GA ran for almost 23 GPU days and is visualised in figure 4.9.

The best individual found, in generation 19, has an accuracy of 82.35%, is 8 layers deep, and has 304,266 parameters. Its genome is shown in listing 4.3.

```
S{C[256(3x3 1x1 S)];C[064(3x3 1x1 S)]}
P MAX[2x2 2x2 V]
P MAX[2x2 2x2 V]
S{C[128(3x3 1x1 S)];C[064(3x3 1x1 S)]}
P AVG[2x2 2x2 V]
P AVG[2x2 2x2 V]
P AVG[2x2 2x2 V]
SMMLP[]
60
```

Listing 4.3: The genome has only 8 layers and is very shallow. It has only two skip layers, spaced out evenly, and pooling layers appear in groups.

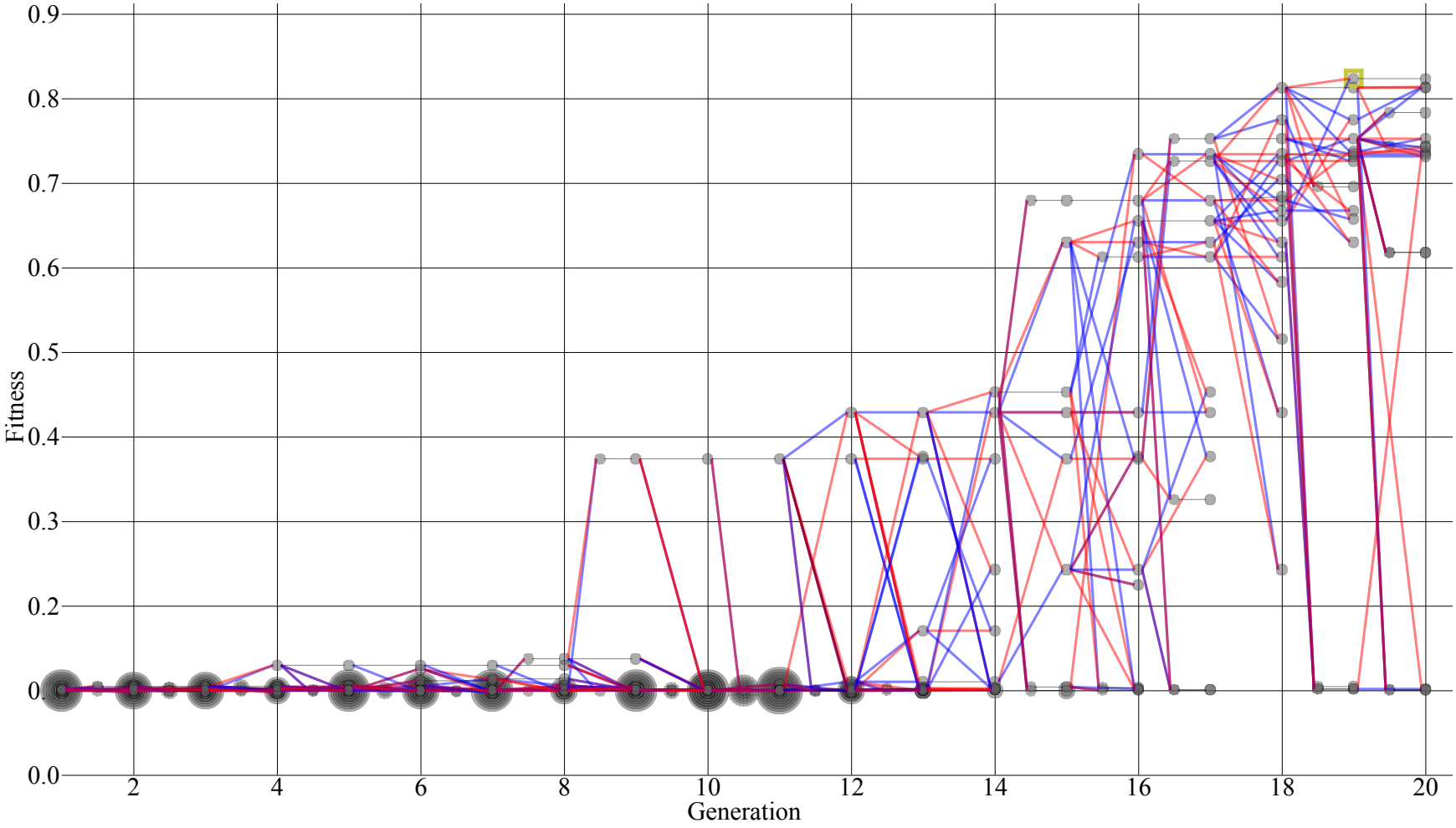


FIGURE 4.9: The first 8 generations all have accuracies corresponding to random chance or slightly above and have few fitness improvements. There are major breakthroughs at generation 9 and 15, both involving cross-over and mutation. In generation 19-20, selection still picks individuals with 10% accuracy for reproduction.

From the GA tree we can see that there were almost no fitness improvements for the first 8 epochs, and most topologies were as good as random chance. There are few but major breakthroughs, and the variety of individuals is quite low until about generation 16. The fitness spread in generation 15 looks similar to the initial generation of our base experiment.

Looking at the depth convergences in figure 4.10, we can clearly see that the initial generation was too deep, but the algorithm recovered from this misconfiguration.

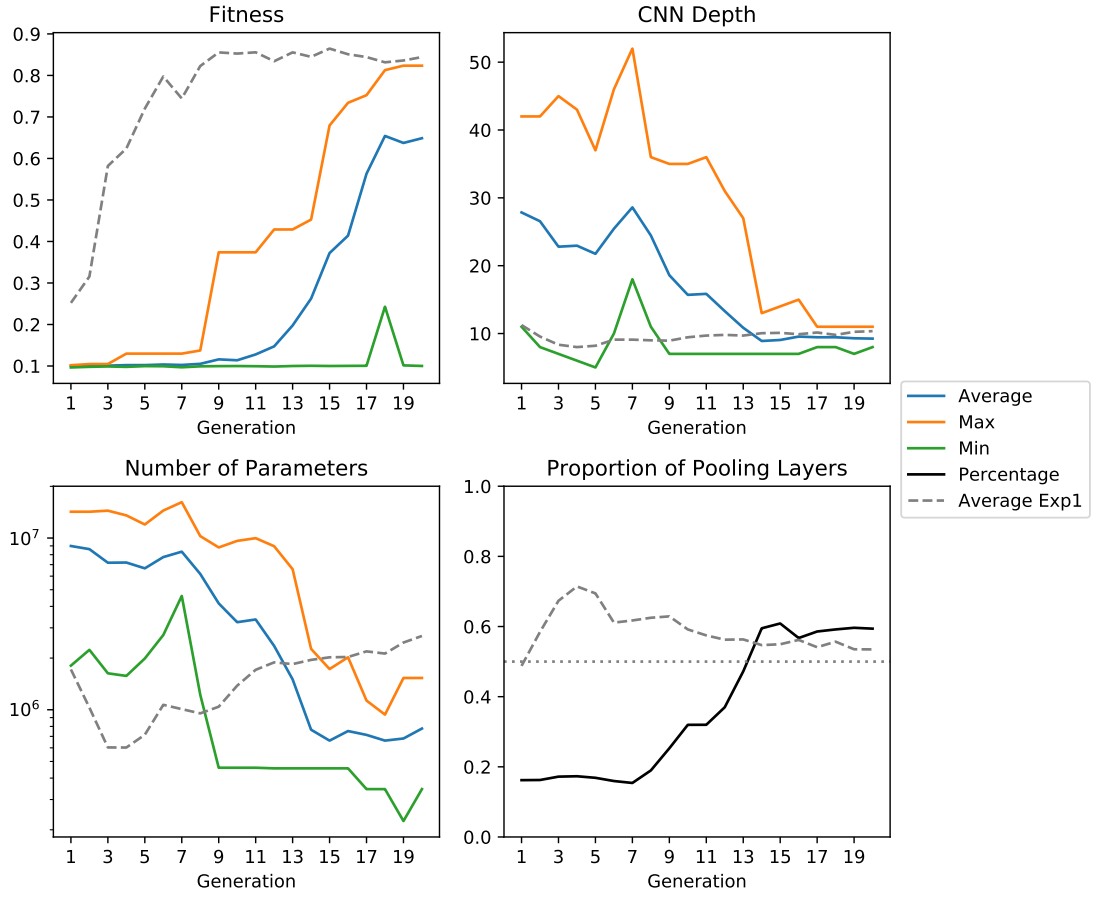


FIGURE 4.10: The fitness converges delayed. The CNNs start much deeper, but converge towards depths comparable to the base experiment. The proportion of pooling layers was clearly set-up incorrectly, but recovered to a similar ratio as our base experiment. In the last generation we contains models with more parameters due to the smaller pooling ratio and bigger depth.

The mean depth converges almost the same depth as the base experiment. In the last two generations, topologies with fitnesses equal to random chance are still selected for reproduction; the selective pressure or variety seems to be too low.

The proportion of pooling layers starts a little under 0.2 and converges up very quickly after generation 7. As shown in section 3.1.2.1, we can have up to five pooling layers without running into half pixels. Because our initial individuals are very deep, the ratio

of pooling layers is proportionately small. This seems to be the wrong approach, because the ratio clearly converges up, and the CNNs get less deep.

The pooling layer ratio, CNN depth, and number of parameters started around the same values as in experiment 2, but in contrast to the previous run cross-over could recover because of correctly spaced out pooling layers.

The main idea of approach 3 was to produce deep networks. The convergence clearly shows that shallow networks were more fit, and the GA was able to converge to the correct depth of roughly the same space as our base experiment. It ending slightly higher is irrelevant as fitness is slightly lower. The fact that in the last two generations of the ancestry tree (4.9) you can see individuals with accuracies equivalent to random chance still being selected for reproduction illustrates that the algorithm took a lot of time to converge into the right parameter space. This can also be seen in figure 4.10 in general. We can say that approach 3 is behind.

This initialisation approach therefore seems to be unfit for our parameter configuration and the data set. It is however important to note that this might change if we were to train individuals for longer. Because the computational load was already quite high, this thesis was not evaluated.

Table 4.3: Transferring the base configuration to MNIST

Parameter	Configuration
Data Set	MNIST
Training epochs	6
Batch size	100
Learning rate decay points	{1, 3, 5}
GPUs	2 Volta V100 Enterprise

4.4 Base Configuration on MNIST

How well does the algorithm adapt to the much easier MNIST data set? For the GA to be of use, it should show similar convergence as the base experiment. If we can show that the algorithm works well on complex and easier tasks, we can hope for it to be of general use.

The base experiment from table 4.1 has been slightly adapted, the changes are shown in table 4.3. The number of training epochs has been brought down to 6 and the learning rate decay points were scaled accordingly. This was done for computational reasons - we have prior knowledge that MNIST requires much less training than CIFAR [12]. This change is probably not impairing the outcome of this experiment as long as accuracies are reasonably high, at least 95% or higher. Lastly, this experiment was run on different hardware for logistic reasons; as we are not comparing wall times to previous experiments, this factor is irrelevant.

4.4.1 Results

The GA ran for 30 GPU hours and is visualised in figure 4.11.

The best individual found, in generation 7, has an accuracy of 99.44%, is 12 layers deep, and has 3.2 million parameters. Its genome is shown in listing 4.4.

```
S{C[128(3x3 1x1 S)];C[128(3x3 1x1 S)]}
P MAX[2x2 2x2 V]
S{C[256(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[256(3x3 1x1 S)];C[256(3x3 1x1 S)]}
S{C[256(3x3 1x1 S)];C[256(3x3 1x1 S)]}
P AVG[2x2 2x2 V]
S{C[064(3x3 1x1 S)];C[128(3x3 1x1 S)]}
P MAX[2x2 2x2 V]
S{C[064(3x3 1x1 S)];C[064(3x3 1x1 S)]}
P AVG[2x2 2x2 V]
P AVG[2x2 2x2 V]
SMMLP[]
6
```

Listing 4.4: Skip and pooling layers are distributed quite evenly. With 12 layers, the network is deep for an MNIST classifier.

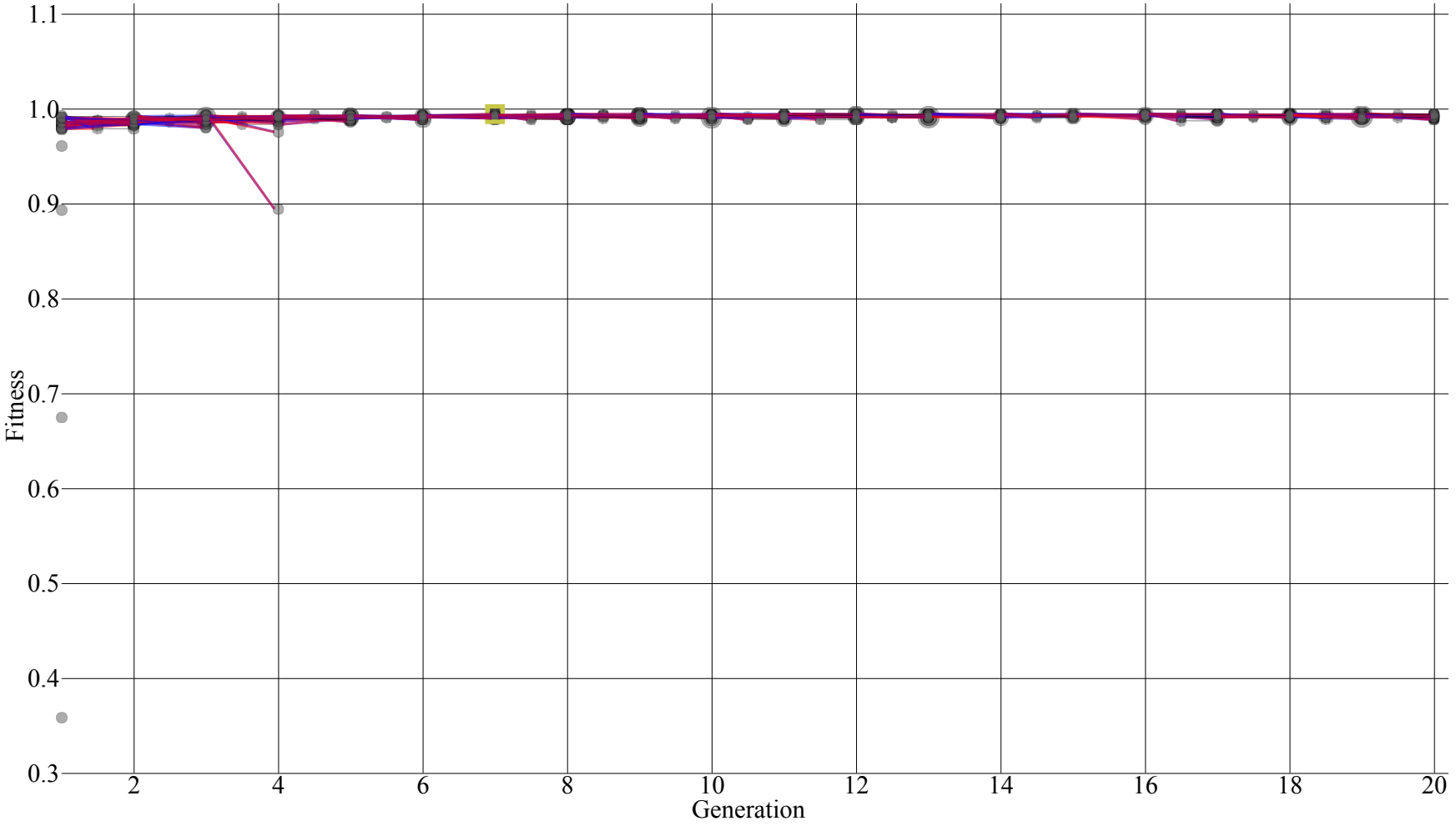


FIGURE 4.11: The GA shows almost no variety, the fitness spread converged to the same value. The initial generation has only four topologies that deviate from the big fittest cluster, and none of those individuals were selected for reproduction. In generation four, one individual is significantly less fit and was not selected for reproduction either.

The GA tree, figure 4.11, shows that the algorithm does not adapt well to the data set. There are four topologies in the initial generation that do not lay within the big fitness cluster at around 0.98. These four topologies were not selected for reproduction. Neither was the one individual breaking out the fitness cluster in generation 4.

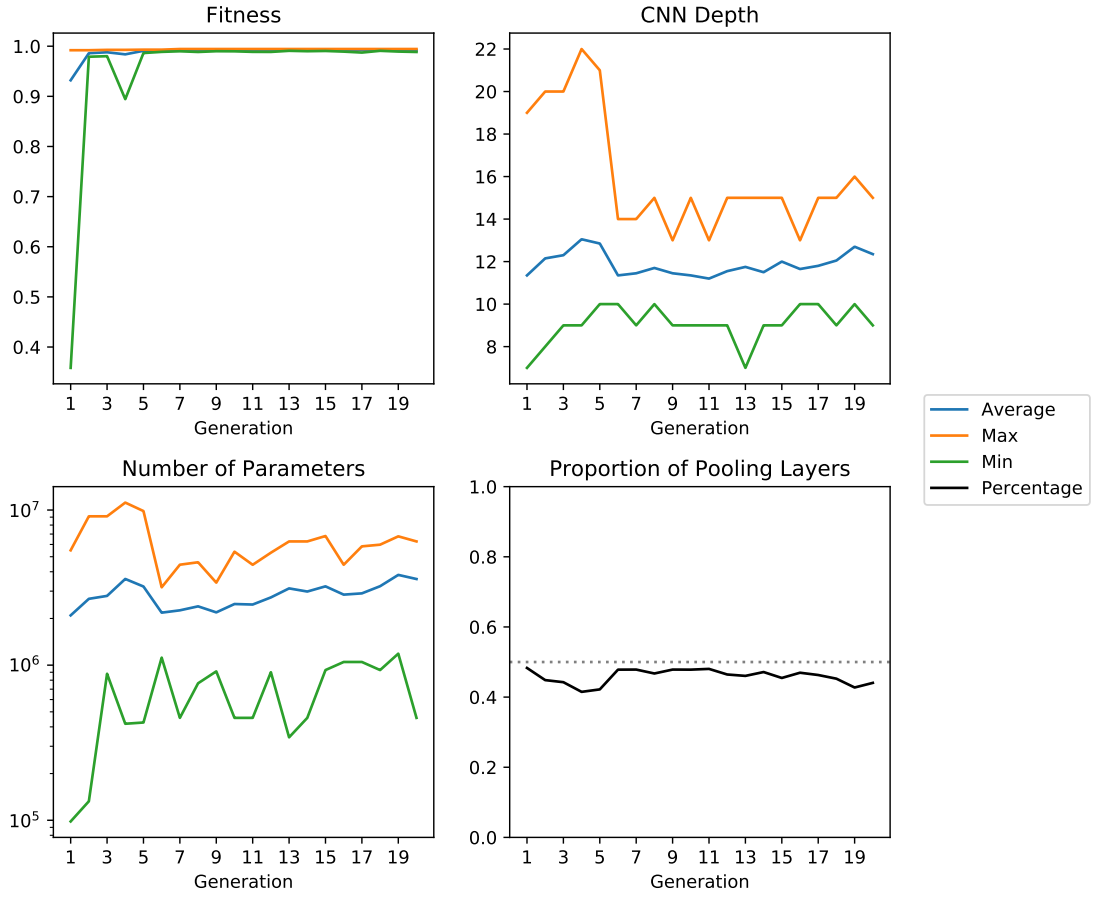


FIGURE 4.12: The mean, average and max fitness almost collapses to one line within 3 generations with one break-out. The CNNs are around 12 layers deep on average, which is deeper than the ones on CIFAR. The number of parameters is proportional to the CNN depth. The ratio of pooling layers is constantly under 0.5

From the GA tree and the fitness curve in figure 4.12 we can clearly say that the selective pressure is too high for this data set. Less fit individuals have no chance of reproduction and the variety of individuals suffers. The min, max and average fitness converged to the same value very early.

As discussed in section 2.2.2, selective pressure can be reduced by increasing the population size or changing the selection algorithm. We are already using a selection algorithm with low selective pressure, but we could increase the population size. Changing population size for all experiments would reduce convergence on the base experiment (section 4.1). Changing pressure specific to this experiment would require us to know the requirement of less selective pressure a-priory.

The CNN depth and number of parameters plummet in generation 6 without impacting the fitness. The reason for this was not found out and might be just random noise. The mean depth of the CNNs found are around 12 which is two layers higher than the base experiment on CIFAR. This is surprising as MNIST networks are normally shallower than CIFAR networks. It shows that the algorithm has almost no gradient to follow.

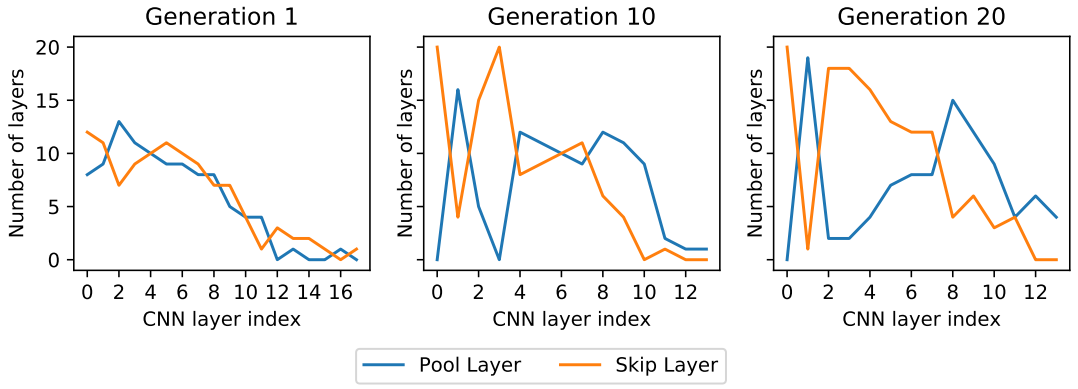


FIGURE 4.13: Similar to the base experiment, the population converges to have more skip layers in the first and more pooling layers in the second half. The pattern of pairwise alternating skip and pooling layers at layers in the first three layers re-occurs, however it was already there in the first generation by random chance.

In figure 4.13 we can see that the layers converged to a similar distribution as the base experiment: By generation 10, the first layer is always a skip layer, and skip/pool layers alternate in the first few layers. The first half prefers skip layers, the second half prefers pool layers. Note that the alternating behaviour can also be observed in the first generation that was created by random chance, however selective pressure clearly increased it.

4.4.2 Comparison to literature

The resulting accuracy of 99.44% is good, but not outstanding. A six layer CNN can achieve accuracies up to 99.73% [22].

4.5 Regularisation

As motivated and described in section 3.3.3, we decrease the fitness by 0.05 per hour, measuring training and testing time. This is supposed to motivate individuals to evaluate faster and trade-off accuracy and evaluation time.

4.5.1 Results

The GA ran for almost 14 GPU days and is visualised in figure 4.15.

The individual with the highest fitness of 0.85 and an accuracy of 88.42% was found in generation 15. The individual with the highest accuracy of 89.06% was found in generation 19 and needed 15 minutes longer to be evaluated. The experiment was 30 hours (8%) faster than the base experiment and yielded an accuracy that is 0.01 better.

```
S{C[256(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[256(3x3 1x1 S)];C[128(3x3 1x1 S)]}
P AVG[2x2 2x2 V]
S{C[128(3x3 1x1 S)];C[128(3x3 1x1 S)]}
P AVG[2x2 2x2 V]
S{C[256(3x3 1x1 S)];C[128(3x3 1x1 S)]}
P AVG[2x2 2x2 V]
P AVG[2x2 2x2 V]
P AVG[2x2 2x2 V]
SMMLP[]
60
```

(A) The genome with highest fitness

```
S{C[256(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[256(3x3 1x1 S)];C[128(3x3 1x1 S)]}
S{C[256(3x3 1x1 S)];C[128(3x3 1x1 S)]}
P AVG[2x2 2x2 V]
P AVG[2x2 2x2 V]
S{C[256(3x3 1x1 S)];C[128(3x3 1x1 S)]}
P AVG[2x2 2x2 V]
P AVG[2x2 2x2 V]
P AVG[2x2 2x2 V]
SMMLP[]
60
```

(B) The genome with highest accuracy

FIGURE 4.14: The two best individuals found differ in the order of a skip/pooling layer in depths 3-4, and the faster individual has 128 instead of 256 filters in that skip layer

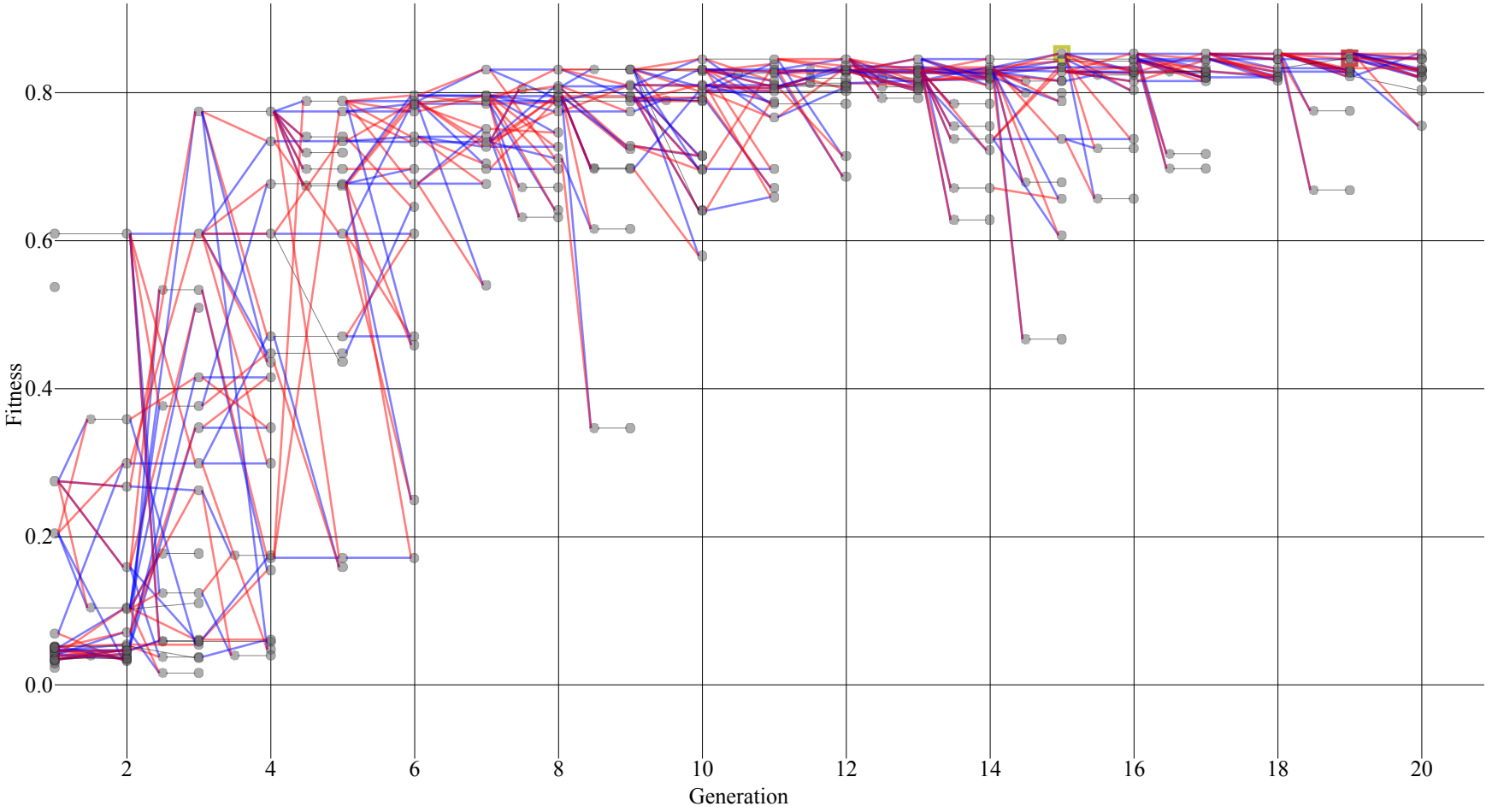


FIGURE 4.15: Two "best" individuals are highlighted: The one with the highest fitness (golden box) in generation 15 and the one with the highest accuracy (red box) in generation 19. The GA tree looks very similar to the base experiment.

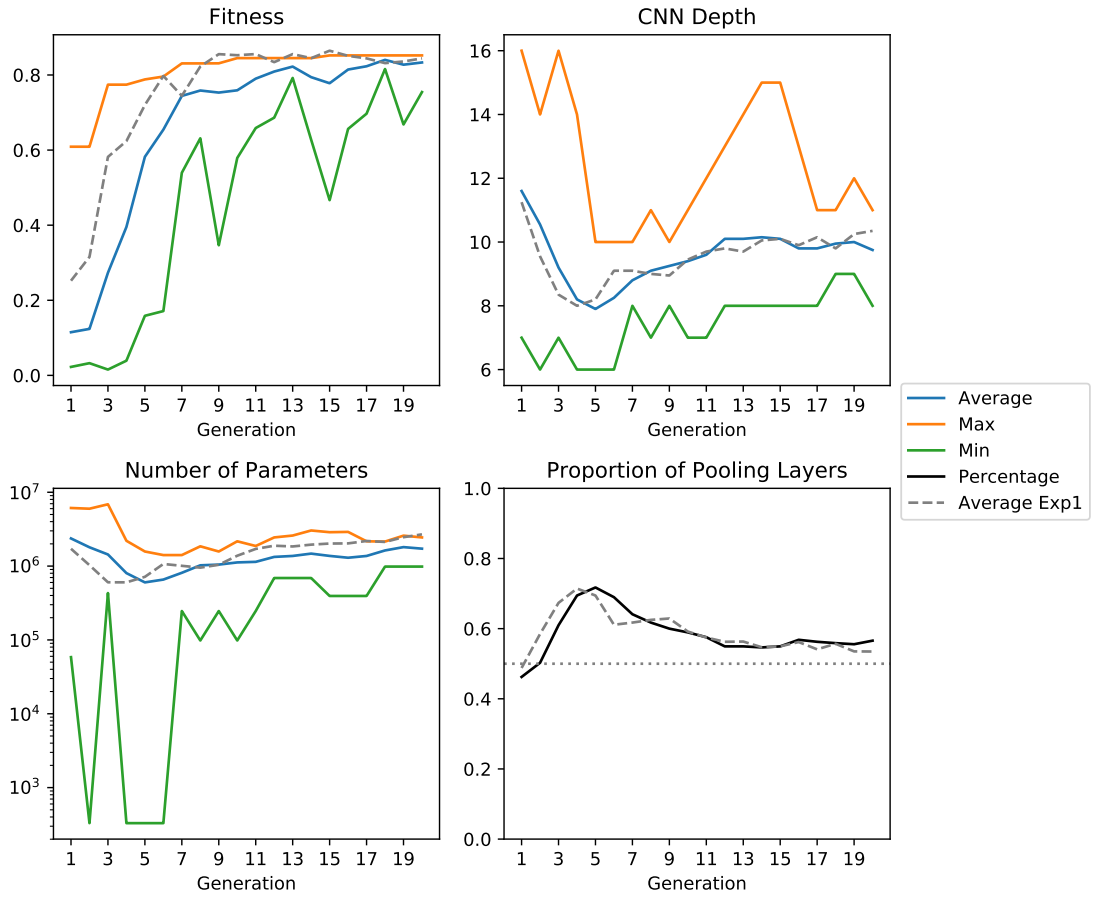


FIGURE 4.16: The networks converge to be 1-2 layers shallower on average. The convergences look very similar to figure 4.3 with no significant differences in overall shape, with a consistent offset on the X axis

Both the GA tree (figure 4.15) and the convergences (figure 4.16) look very similar to the base experiment. Networks are 1-2 layers shallower on average, but the top accuracy found is even 0.01 better. Note that the graph shows fitnesses, not accuracies. The algorithm therefore sorted out deep networks that performed as good as shallower networks successfully.

Figure 4.16 shows a constant, short delay on the X axis. This delay occurs on all graphs. Because through random chance the first generation is offset on the Y axis into the direction opposing convergence, and the first generation is independent of regularisation, the offset on the X axis could just be an effect of the algorithm needing a generation more to mitigate this Y offset.

The regularised run favoured having less filters per skip layer. Figure 4.17 shows that instead of converging towards a majority of 256 filters, it was advantageous to have 128 filters. This indicates that having less filters increases training speed and maintains accuracy. At the same time, the kernel functions evolved to the average instead of the max function. Why this happened is hard to say because tuning the kernel function manually is always done per trial and error.

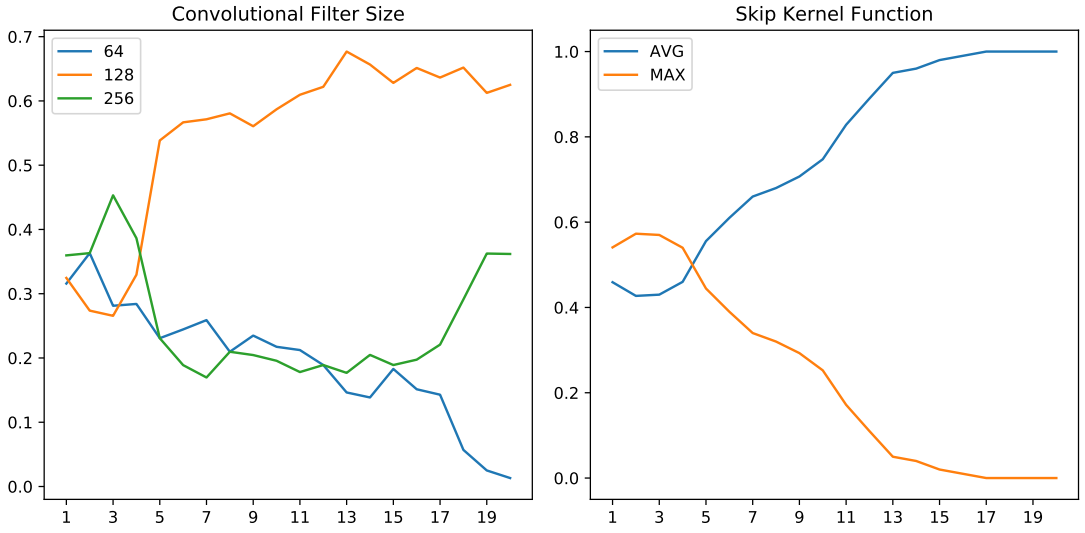


FIGURE 4.17

The two topologies found (listing 4.14) are are very similar. They differ only in the order of layer 3 and 4, and in the number of filters in one of the skip layers. The faster individual uses a filter size of 128 instead of 256 filters in one instance. This small change allowed the individual to evaluate with 15 minutes difference and only sacrificed 0.0064 accuracy. Such small changes add up significantly in the course of the GA and allowed it to evaluate 8% faster than our base experiment without sacrificing accuracy of the end result.

4.6 Partial Training with Linear Epoch Function

As described in section 3.2.3, we extend our base experiment from using a constant number of 60 epochs to a linear function. This function depends on the generation index and goes from a lower to an upper bound. We set the lower bound to 30, which is half the number of epochs in the base experiment, and the upper bound to 70 which is a bit higher than the base experiment; see figure 4.18. This function was chosen to ideally make the algorithm find better topologies than the base experiment in less time.

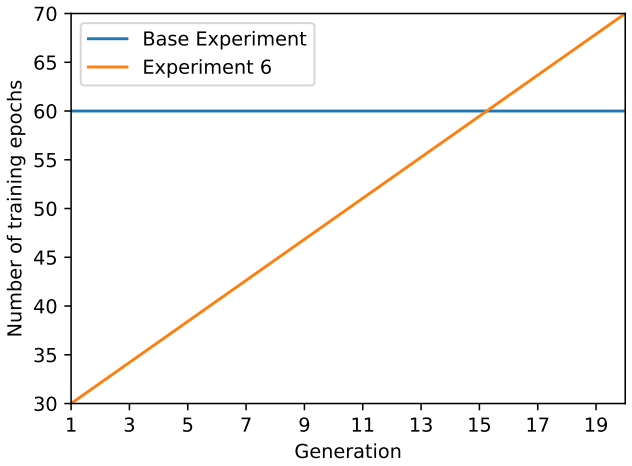


FIGURE 4.18: The number of epochs is a linear function that starts at half the base experiment and ends a bit higher, supposed to return better CNNs in less time.

4.6.1 Results

The GA ran for a bit over 12 GPU days and is visualised in figure 4.19.

The best individual found, in generation 18, has an accuracy of 88.7%, is 9 layers deep, and has 1.6 million parameters. Its genome is shown in listing 4.5.

```
S{C[128(3x3 1x1 S)];C[064(3x3 1x1 S)]}
S{C[128(3x3 1x1 S)];C[256(3x3 1x1 S)]}
P MAX[2x2 2x2 V]
P AVG[2x2 2x2 V]
S{C[256(3x3 1x1 S)];C[256(3x3 1x1 S)]}
P AVG[2x2 2x2 V]
P AVG[2x2 2x2 V]
P AVG[2x2 2x2 V]
SMMLP[]
64
```

LISTING 4.5: There are only three skip layers, one less than the topology found by the base experiment, but it performs just as well.

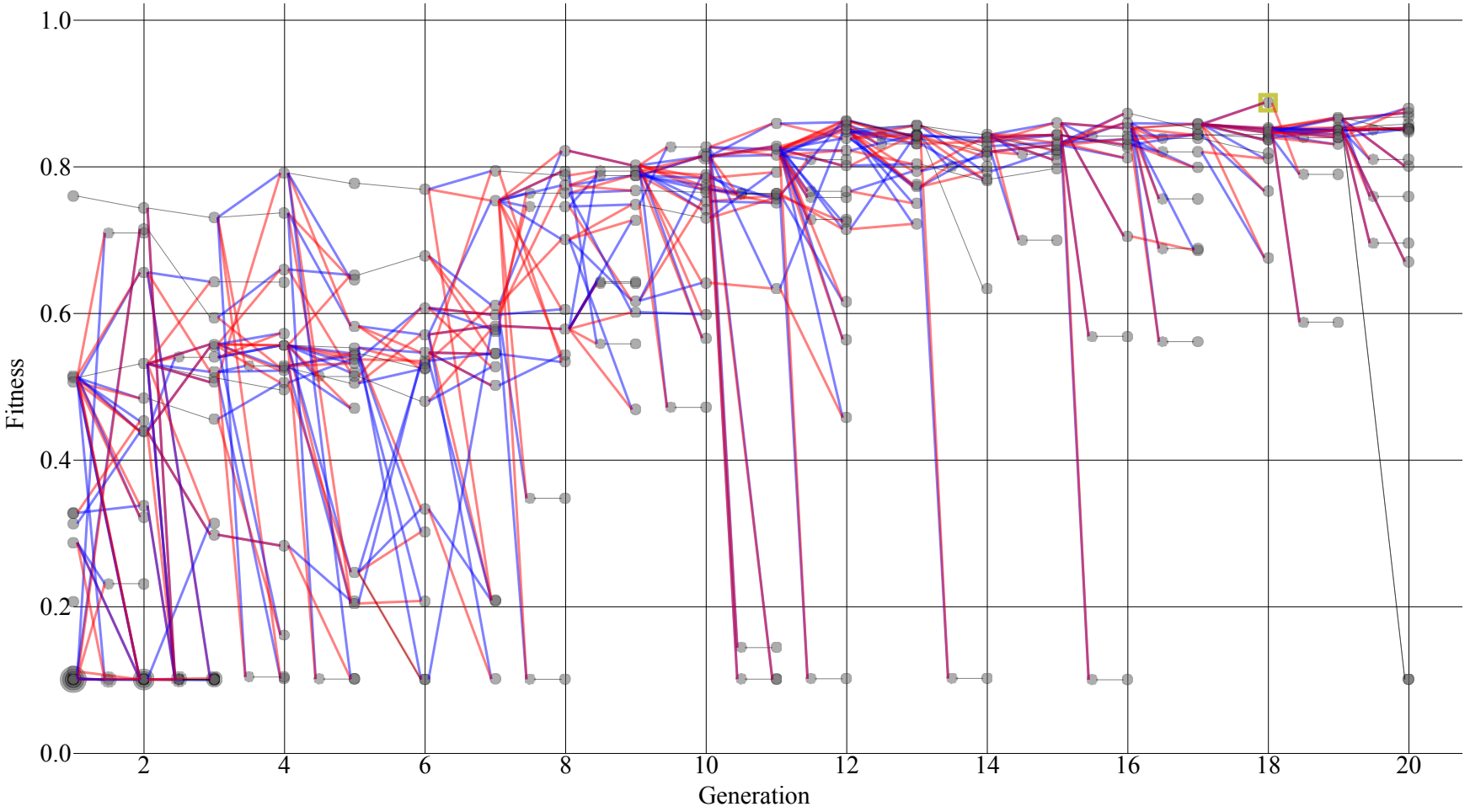


FIGURE 4.19: The GA converges slower than the base experiment and there is a bit more variety. When elitism copies the best topology and the individual is trained further, the accuracy often goes down (i.e. generations 1-3), but not always (generation 3 to 4).

The GA tree, figure 4.19, converges slower and more gradually than the base experiment. This is not a drawback since we are training them for shorter durations in the beginning, so reluctant fitness convergence is one of the core features of this algorithm. This reluctant convergence brings a bit more variety in fitness evaluations.

The individuals that resulted from an elitist clone often lose fitness, for example the best individual in generations 1-3. Our elitism implementation clones the topology into the new generation before the linear epoch function is applied to the population. This means that the previously best individual will also be trained further and therefore change its score. If the elitist individual loses fitness, it indicates that the model is overfitting to the training set, and training it further reduces the accuracy on unseen data. This would mean that in the previous generation, the individual was on the edge of overfitting. We want the algorithm to decrease the fitness at this stage and not take the higher previous value, because it might find another individual that overfits less. By punishing the overfitting individual, we allow others to be selected more likely. The algorithm is effectively sorting out topologies that overfit and from time to time, such as in generation 4, a new individual is found that works better for the current number of epochs. This is another factor to explain slower convergence and higher variety: The selective pressure goes down because individuals, even the best individual, lose fitness over time.

Figure 4.20 illustrates slower convergence. The overall movements are similar to the base experiment, but they are delayed and stretched out, for example the CNN depth stretches for twice the duration on X axis.

So far "slower convergence" was measured in terms of number of generations, not in actual wall time. Figure 4.21 illustrates the speed-up in hours, compared to the base experiment.

We can see that the speed-up is anti-proportionate to figure 4.18 with some noise, this is as expected. The algorithm converged much faster than the base experiment in actual wall time. The fact that the first two generations were evaluated 18 hours faster while maintaining the mean fitness of the base experiment (shown in figure 4.20) shows that the idea of training less in the beginning works very well. By generation 18 we have the same mean fitness as the base experiment despite having saved more than 60 hours of computation.

The run was 70 hours (19%) faster than the base experiment and the best individual found has an accuracy that is a little bit (0.0034) higher. These results strongly indicate that this method is a viable approach to reduce training time.

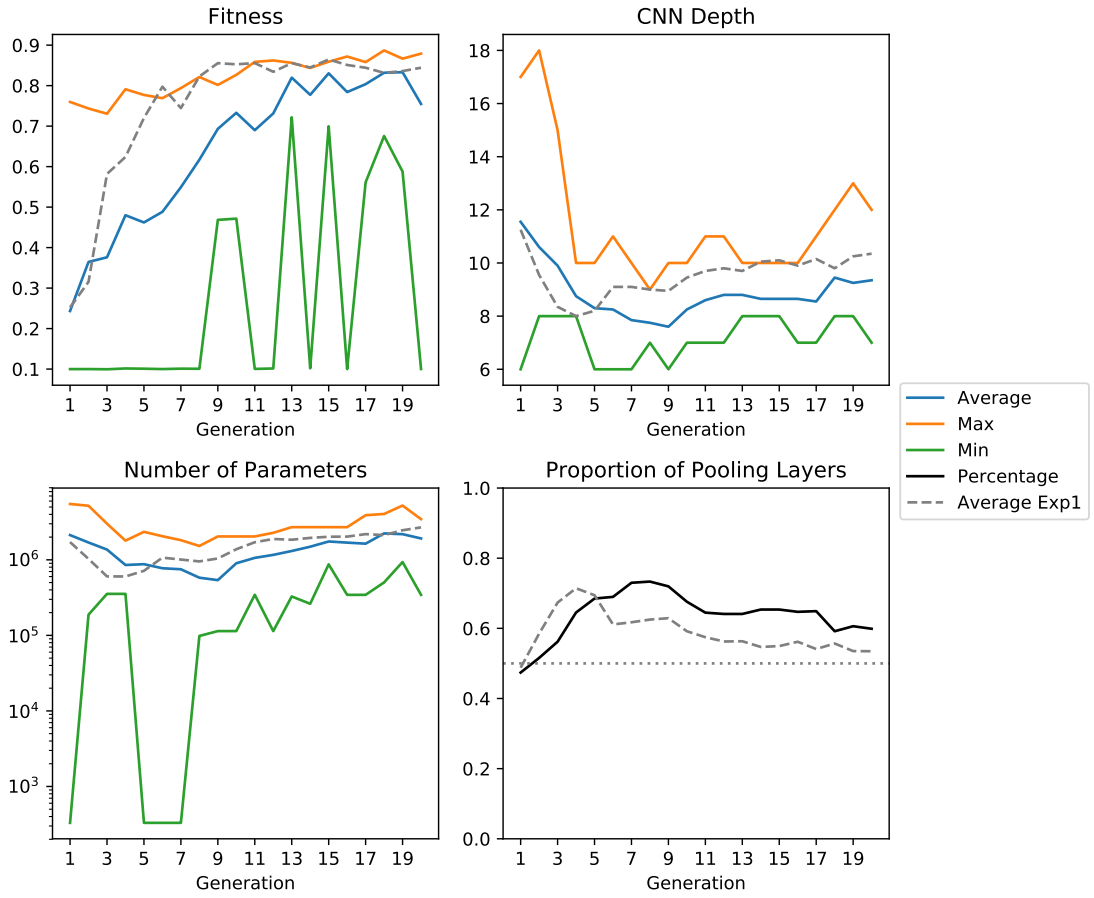


FIGURE 4.20: The algorithm converges slower in respect to generations, but it reaches the same mean accuracy as the base experiment in generation 18. The depth converges slower than the base experiment, and from generation 9 on it has an offset on the Y axis.

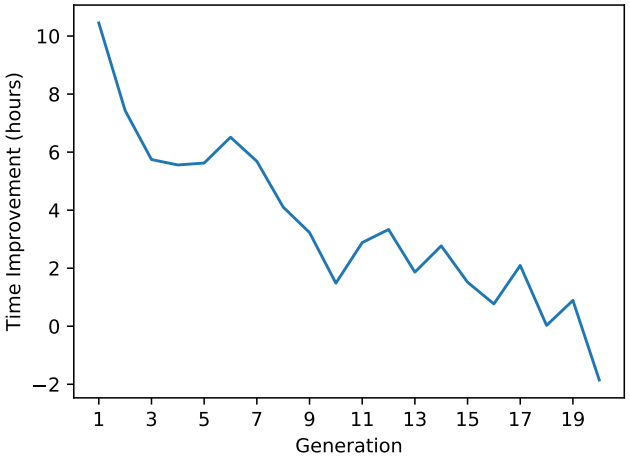


FIGURE 4.21: Earlier generations evaluated much faster, later generations take longer time than the base experiment

Chapter 5

Conclusions

We have discussed and analysed how to evolve convolutional neural network topologies for image recognition using genetic algorithms. During the in-depth discussion of the algorithms used, a fundamental challenge, the "half pixel problem", was described and solved, resulting in three CNN initialisation approaches. Approach 2 was shown to be completely unfit. The first approach worked better for the experiments that we conducted, but we noted that approach 3 might work better for deeper networks and longer training. This needs to be evaluated further.

The genomic representation that was introduced allows to document the topologies quickly and precisely, while still working well for cross-over and mutation. A novel GA visualisation technique gave insights into the ancestry of individuals and helped explaining how the different techniques influenced the population.

During the six experiments conducted, a common design pattern of alternating pooling and skip layers was repeatedly found by the evolutionary process. We observed this pattern to occur pairwise, why that is and if it is beneficial is yet to be explored. The GA correctly converged to have more pooling than skip layers in the first part of the genome, and the inverse later. Other measures of complexity and layer distributions converged towards an optimum throughout our experiments. These results clearly show that the algorithm does indeed design the CNN topologies correctly and "intelligently".

Our base experiment converged to the biggest number of 256 filters within its skip layers, regularised models tended to favour 128 filters instead as a trade-off between time and accuracy. Regularisation was able to speed up the process by 9% while maintaining result performance.

The CNN topologies found throughout the experiments were good, but not outstandingly accurate. Most notably, the topologies were not very deep. We suspect the reduced training duration to be the root cause of this as we have decreased the number of epochs substantially to reduce wall time. Our assumption that "deeper is better" was disproved

by the algorithm, at least for the search space that we imposed. However, even when we forced the algorithm to have deep networks, it was able to adapt and recover well (for the two working initialisation approaches), converging to the usually observed depth.

The core problem, how to speed up the process, was explored by reintroducing partial training. We were able to bring down wall time by 19% without an impact in fitness, and even though we measured slower convergences relative to the number of epochs, the algorithm converged much quicker relative to wall time. We found that most inferior topologies in early generations can be sorted out without training for long, and successive partial training is a good strategy to do so. A lot of best models were on the edge to overfitting, and training them further decreased their fitness and allowed other individuals to be selected for reproduction more likely.

On the MNIST data set, the selective pressure was shown to be too high, reducing the variety and stagnating the algorithm. This is a worrying problem as it impairs the core idea of it working out-of-the-box for unknown data sets without extensive configuration and domain knowledge. Further research needs to compare a variety of data sets with different levels of complexity and adapt selective pressure automatically, for example by measuring fitness spread and adjusting the population size.

A related research frontier is how to set up machine learning and evolutionary hyper parameters truly autonomously. So far, the number of training epochs, momentum, learning rate decay points, etc. have been set a-priori involving domain knowledge. Especially the number of epochs is very specific to the difficulty of the data set and knowing it a-priory for novel data is unlikely and would require a lot of experience.

Acronyms

BNF Backus-Naur Form.

CNN Convolutional Neural Network.

GA Genetic Algorithm.

GPU Graphical Processing Unit.

MLP Multilayer Perceptron.

SMMLP Softmax Multilayer Perceptron.

Glossary

Fitness A measure of expected reproduction.

Generation A population at a point in time.

Genome An alpha-numerical representation of an individual that is sufficient for calculating fitness.

Hill Climber Local search algorithm that tries out incremental changes to a configuration until it gets stuck on a local optimum.

Multilayer Perceptron A feed-forward neural network with optional hidden layers.

Population A set of genomes.

Same Padding Adds fillers to the input data (classically zeroes) in order to stop (convolutional) layers to change the size of the data stream.

Selective Pressure How much the algorithm favours individuals of high fitness for reproduction. Or: how unlikely it is that a bad individual is picked for reproduction.

Softmax Multilayer Perceptron A classifier that uses the softmax function on the output layer of a multilayer perceptron, used as the last step in our CNNs.

Unit filter / Unit kernel / Unit stride A filter / kernel / stride of 1x1.

Valid Padding Not using same padding.

Appendix A

Code

The code to this project has been handed in to the University of Southampton as a design archive.

A.1 Table of Contents

- cga_src.py: Implements all GA core functionality
  - base.py: All abstract and base classes
  - cnn.py: The CNN class, history, pickling
  - convolutional_layer.py
  - data.py: The data sets
  - evaluator.py: Objective fitness function
  - export.py: Diagram exporter for GA ancestry tree visualisation
  - ga.py: The genetic algorithm
  - hardware.py: Manages async GPUs
  - kernel.py: All kernel maths
  - pooling_layer.py
  - population.py: Caches individuals, fitnesses, implements selection
  - skip_layer.py
  - softmax_mlp.py
- experiments: The six pickled experiments conducted are over 40 GBs and not included
- tests: Unit tests to ensure GA functionality

- visualise:
  - visualise.py: Creates the ancestry tree visualisation by parsing the pickles into yaml files
  - frontend: The interactive web interface
    - data: Holds parsed yaml files to be read by JavaScript
    - lib: Third party dependencies
    - available_data.js: Holds references to all yamls for frontend, generated automatically
    - main.js: Renders the yaml files interactively
  - graphs.py: All graphical analysis except for GA tree
  - lin_epoch.py: The visualisation of the linear epoch function
- cga.py: The main import for the GA framework
- ga.*.py: Python script for the experiments
- iridis.*.sh: Iridis wrapper for python scripts
- pickling_ga.py: Helper script for starting experiments
- sync.sh: Synchronisation script for me to fetch results with

A.2 Experiment matching

The experiments (pickles, yaml files, ...) are named more concisely in the archieve than in the thesis. This is the mapping from experiment number to file name:

1. data-CIFAR10__popsize-20__crossover-0.9__mutation-0.2__punishment-per-hour-0__learning-0.1_0.9_[1, 26, 43]_0.9__epochfn-const_epochs_60
2. approach-2__data-CIFAR10__popsize-20__crossover-0.9__mutation-0.2__punishment-per-hour-0__learning-0.1_0.9_[1, 26, 43]_0.9__epochfn-const_epochs_60
3. approach-3__data-CIFAR10__popsize-20__crossover-0.9__mutation-0.2__punishment-per-hour-0__learning-0.1_0.9_[1, 26, 43]_0.9__epochfn-const_epochs_60
4. data-MNIST__popsize-20__crossover-0.9__mutation-0.2__punishment-per-hour-0__learning-0.1_0.9_[1, 3, 5]_0.9__epochfn-const_epochs_6'
5. data-CIFAR10__popsize-20__crossover-0.9__mutation-0.2__punishment-per-hour-0.05__learning-0.1_0.9_[1, 26, 43]_0.9__epochfn-const_epochs_60
6. data-CIFAR10__popsize-20__crossover-0.9__mutation-0.2__punishment-per-hour-0__learning-0.1_0.9_[1, 30, 50]_0.9__epochfn-linear_epochs_30_70

A.3 Technical Setup

A.3.1 Install

We suggest conda to manage dependencies.

```
conda env create environment.yml
conda activate cnn
# or
pip install -r requirements.txt
```

A.3.2 Run tests

The unit tests cover all GA functionality.

```
python -m unittest discover tests/
```

A.3.3 Run experiment

Tun the python file. The experiments will automatically try to open a pickle from the experiment folder. As the experiments are bigger than 40GBs, they were not handed in, and the GA will start at generation 0.

```
python ga.cifar.approach3.px
```

A.3.4 Run the ancestry visualisation

```
cd visualise
# Rebuild the yaml files for the diagram (only needed when experiments change)
python visualise.py

# Start the web server to visualise
python server.py
```

Or, head to https://ga.yaron-strauch.com

Bibliography

[1] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[2] Jake Bouvrie. Notes on convolutional neural networks. 2006.

[3] Dan Cireşan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. *arXiv preprint arXiv:1202.2745*, 2012.

[4] Lawrence Davis. Handbook of genetic algorithms. 1991.

[5] Kenneth A De Jong. *Evolutionary computation: a unified approach*. MIT press, 2006.

[6] Kenneth A De Jong and William M Spears. Using genetic algorithms to solve np-complete problems. In *ICGA*, pages 124–132, 1989.

[7] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[8] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc.", 2017.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[10] Gregory Hornby, Al Globus, Derek Linden, and Jason Lohn. Automated antenna design with evolutionary algorithms. In *Space 2006*, page 7242. 2006.

[11] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[12] mridul joshi. An introduction to cnns and a step by step model of a digit recognizer using mnist database in python, 2018. URL `https://medium.com/coinmonks/an-introduction-to-cnns-and-a-step-by-step-model-of-a-digit-recognizer-using-mnist-database-in-f4ea6af06d77`. Accessed August 2019.

[13] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[15] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[16] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[17] Brad L Miller, David E Goldberg, et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.

[18] Susan M. Mooney. H. j. muller and r. a. fisher on the evolutionary significance of sex. *Journal of the History of Biology*, 28(1):133–149, Mar 1995. ISSN 1573-0387. doi: 10.1007/BF01061249.

[19] Peter Naur, John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John McCarthy, and Alan J Perlis. *Revised report on the algorithmic language Algol 60*. Association for Computing Machinery, 1976.

[20] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[21] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2902–2911. JMLR. org, 2017.

[22] Vadim Romanuke. Parallel computing center (khmelnitskiy, ukraine) gives a single convolutional neural network performing on mnist at 0.27 percent error rate. URL `https://drive.google.com/file/d/0B1WkCFOvGHDdOC0yR0tfbmpidjg/view?usp=sharing`. Accessed June 2019.

[23] Sumit Saha. A comprehensive guide to convolutional neural networks — the eli5 way, 2018. URL `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`. Accessed August 2019.

[24] Patrice Y Simard, David Steinkraus, John C Platt, et al. Best practices for convolutional neural networks applied to visual document analysis. In *Icdar*, volume 3, 2003.

[25] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[26] Yanan Sun, Bing Xue, Mengjie Zhang, and Gary G Yen. Automatically designing cnn architectures using genetic algorithm for image classification. *arXiv preprint arXiv:1808.03818*, 2018.

[27] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[28] Darrell Whitley. An overview of evolutionary algorithms: practical issues and common pitfalls. *Information and software technology*, 43(14):817–831, 2001.

[29] Lingxi Xie and Alan Yuille. Genetic cnn. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1379–1388, 2017.