EVOLVING HIERARCHICAL STRUCTURES FOR CONVOLUTIONAL NEURAL NETWORKS USING JAGGED ARRAYS

by

Stuart McIlroy

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
March 2018

Table of Contents

List of Tables

List of Figures

Abstract

Traditionally, deep learning practitioners have relied on heuristics and past high-performing networks to hand-craft their neural network architectures because architecture search algorithms were too computationally intensive. However, with the advent of better GPU processing for neural networks, architecture search is now feasible. We propose a search through the architecture space, using a genetic algorithm that evolves convolutional neural networks using a novel decomposition scheme of nested hierarchical structures using jagged arrays. We test it on standard image classification benchmarks (MNIST, CIFAR-10) as well as a modular task (classifying "what" and "where"). The resulting architectures achieve performance comparable with the state of the art. We then explore the types of network architectures that arise from the evolutionary stage. We show that the evolved architectures adapt to the specific dataset, that common heuristics such as modular reuse evolve independently in our approach and hierarchical structures develop for the modular task.

List of Abbreviations Used

CE Cross-entropy.

CIFAR-10 (Canadian Institute For Advanced Research dataset of color images.

CNN Convolutional neural network.

CPU Central processing unit.

EA Evolutionary Algorithms.

FC Fully connected.

GA Genetic Algorithm.

GPU Graphics Processor Unit.

MLP Multi-layered Perceptron.

MNIST Modified National Institute of Standards and Technology database of handwritten digits.

Acknowledgements

I would like to sincerely thank my parents for supporting me throughout this Master's. Without them, I wouldn't be here today.

I would like to thank Dr. Thomas Trappenberg for providing guidance, insightful comments and always pushing me to go further with my research. I have become a more independent and well-rounded researcher as a result.

Chapter 1

Introduction

The popularity of convolutional neural networks (CNNs) has exploded in recent years due to their success on many difficult tasks such as image classification [24], speech recognition [15], machine translation [9] and game playing [41]. Convolutional neural networks are a type of neural network and are part of the Deep Learning family of image representation learning algorithms. Deep learning encompasses many forms of neural networks like feed-forward and recurrent networks that aim to learn a compact, generalizable representation of the data.

Even with the success that CNNs have seen, there remain issues that require special attention to any practitioner that wishes to use them. For example, datasets can be very different from one another. Datasets may be small or large, or feature vastly different objects within the images, or the datasets may be noisy or unbalanced. As per the No Free Lunch Theorem [40], no algorithm can perform optimally on all problems, therefore care must be taken to select the best algorithm given the dataset. This is no different when determining the type of CNN to use to classify a dataset.

Additionally, if the capacity of the network i.e. the number of learnable parameters, is larger than the dataset truly requires, than the network is unnecessarily large — an important consideration in embedded and mobile systems. Also, the complexity of the network may allow it to overfit the training data whereby it memorizes the noise in the training set and lacks the ability to generalize to unseen data. On the other hand, if the network is too small, it may lack the ability to fit the current dataset correctly and have poor accuracy, known as underfitting.

Therefore, most CNNs were traditionally handcrafted and the best performing networks in the literature were designed through trial and error to fit various datasets. However, these handcrafted networks have not been designed in a rigorous manner with systematic parameter search or model-based searching methods, (we may refer to specific architectures and parameter distributions of a neural network as a model).

There always would remain concerns that the particular network that was chosen could be improved.

There has been much work performed on parameter selection of neural networks and also other means such as employing evolutionary algorithms to search the architecture and parameter space to optimize the networks. However, up to recent times, evolutionary algorithms were difficult to use with CNNs. It is not uncommon for models to take weeks to train. Additionally, CNNs tended to produce excellent results without much fine tuning i.e. different models produce similar results. Circumstances changed recently with the advent of faster Graphical Processor Units (GPUs) and the desire by practitioners to squeeze every last drop of accuracy out of models to achieve the very best performance.

In fact, one of the major driving forces for the adoption of CNNs was the creation of algorithms for CNN computation on GPUs which allowed CNNs to train much faster. Researchers are even going so far as to design specialized hardware for Deep Learning models [21]. Therefore, research is now being conducted on utilizing evolutionary algorithms to evolve CNN architectures — an approach that was intractable six years ago.

1.1 Research Statement

Selection of the best performing network architecture is a challenge that is encountered when using convolutional neural networks. New approaches to finding the best performing network architecture for a given problem and gaining a better understanding of the types of neural networks that result from evolution are needed to better leverage the vast computational potential of convolutional neural networks.

1.2 Thesis Overview

We implement a genetic algorithm with novel mutation and crossover operators to evolve neural network architectures in a nested hierarchical manner. The basic building blocks are nested jagged arrays of convolutional and fully connected layers. We test our approach with three datasets: MNIST, CIFAR-10 and a dataset with a two class problem. We achieve competitive performance on MNIST and CIFAR-10 and

successfully evolve a hierarchical structure on the two class problem.

1. We create a novel decomposition scheme of convolutional neural networks using nested structures of jagged arrays. We evolve the structures using a genetic algorithm. The goal is to create a hierarchical model of a CNN which is fit to the problem at hand. The difficulty of this stage was transforming the model into a neural network suitable for training on the GPU.

2. We run the genetic algorithm on three sets of data, MNIST, CIFAR-10 and our 2-class problem. The goal is to test the accuracy of our technique. The difficulty of this stage was integrating the datasets into the genetic algorithm.

3. We observe what structures develop and their distributions among the evolved population of networks. The goal is to investigate the types of networks that are successful and their properties.

1.3 Thesis Contribution

In this thesis, we apply a genetic algorithm to evolve a network architecture and observe the types of networks that result. In particular we make the following contributions:

1. We propose a novel hierarchical decomposition scheme using jagged arrays to represent a neural network. We use novel mutation and crossover operators to evolve the neural networks. This two-step approach allows practitioners and researchers to automatically search for a network architecture that best suits the problem.

2. For the MNIST dataset, we observed that the best networks were flat (non-hierarchical) and short (3 layers). For the CIFAR-10 dataset we found that flat, longer networks performed better than shorter networks, modules are re-used across the networks, the mutations which favor growth are most prevalent and the number of parameters of a network are not necessarily indicative of the network's performance.

3. We observed that hierarchical models evolved for the two class problem and that two separate channels evolved for each of the two classes.

4. We observe that some of the heuristics used today (such as modular design and reuse) are independently evolved and that the networks evolved to suit the problem.

1.4 Organization of the Thesis

The remainder of this thesis is organized as follows: Chapter 2 provides an overview of machine learning, specifically CNNs and genetic algorithms and typical network architectures of CNNs. Chapter 3 presents our approach to evolving network architectures. Chapter 4 presents the results and provides a discussion of our results. Chapter 5 concludes the thesis, presents the limitations and threats to validity and proposes possible avenues for future work.

Chapter 2

Background and Related Work

This section presents an overview of machine learning and more specifically, convolutional neural networks.

2.1 Machine learning

The goal of machine learning is to improve performance on a *task* given *experience*. The performance is measured using a specific *performance measure*. The *model* is a representation of the task and should perform better with more experience. Generally, the experience is presented in the form of a collection of $\mathbb{R}^n$ vectors known as data. The model should perform well with data that the machine learning algorithm has never seen before. This is called generalization.

Supervised learning's goal is to learn a correct mapping from X to y where X is a collection of $\mathbb{R}^n$ vectors of data and y is an accompanying vector $\{1, ..k\}$ or $\mathbb{R}^n$. X and y can be discrete or continuous valued numbers. A training set X_{train} is provided that comes with labeled data. So that each x data points has an accompanying y value. The learning algorithm must learn the mapping of training data x to y with the added goal of generalizing to unseen examples of data on a test set X_{test}. Supervised learning can further be divided into classification and regression tasks. Classification tasks are where the y label is discrete and the goal is to label each x instance with one of the possible y values. In regression tasks, the goal is to select a real value y for each instance x. In this thesis, we focus on supervised learning.

2.1.1 Overfitting and Under-fitting

Overfitting occurs when an algorithm performs well on the training set but performs poorly on a test set. As stated earlier, the goal is to perform well on both the training and test set. Overfitting occurs because the algorithm may have learned from random noise or outliers in the training set which do not occur in the test set or the algorithm

has only learned to memorize the training set rather than learn to generalize to as yet unseen data. Under-fitting occurs when the data performs poorly on the training set due to insufficient training data or the algorithm's capacity is not large enough to handle the training data's complexity.

Capacity is a measure of the ability of an algorithm to fit many functions. In other words, it is a measure of the maximum complexity of functions that an algorithm can fit. The larger the capacity, the more complex the function that the algorithm can fit. A rough estimation of capacity can be made by determining the number of free learnable parameters the algorithm contains during training.

Hypothesis space is a related concept and can be thought of as the number of functions that the machine learning algorithm has at its disposal to fit the problem. No single machine learning algorithm can perform well on every single problem as per the No Free Lunch Theorem [40]. Therefore, the challenge remains to find the right algorithm for the right task.

2.1.2 Variance and bias trade-off

Bias is the measure of average error in the training data. Variance is the measure of how well an algorithm performs across many constructions of the same model with a different sample of the dataset. A very complex model will have a low bias, but a high variance. A simple model will have a high bias but a low variance. For an optimal algorithm, the goal is to construct a model with low variance and low bias but in reality, most algorithms settle for a trade-off between the two. Neural networks generally have a high variance and a low bias.

2.1.3 Neural network

Neural networks are learning algorithms inspired by how the neurons in our brains work. Indeed, neural networks can be seen as abstract representations of networks of neurons. They were originally devised by Rosenblatt in 1958 and called a perceptron [38]. A perceptron is a set of neurons which are fed input from the data and output a classification value. It was a very simple implementation that lacked many modern day features such as backpropogation but performed well enough to solve simple linearly separable problems.

A single threshold neuron has n incoming weights which each correspond to an input from a data example $\boldsymbol{x}$. The weights are multiplied by the input $\boldsymbol{w} \times \boldsymbol{x}$ and summed up. If the value is greater than a bias then the output is 1 otherwise it is 0. The bias is a parameter of the neural network. It can be thought of as the threshold which the weights times the outputs must reach before the neuron fires.

$$f(x) = \begin{cases} 1, & \text{if } w \cdot x + b > 0 \\ 0, & \text{otherwise} \end{cases}$$

Threshold perceptrons are able to classify linearly separable data, but were deemed unsuitable for complex problems due to their inability to solve non-linear problems such as the XOR problem [32]. Afterwards, Werbos invented the backpropagation technique which solved the XOR problem using a multi-layer perceptron [37]. In the 1980s, Rumelhart et. al. popularized the technique under the term connectionism [39].

Multi-layer perceptrons (MLP) have several differences to perceptrons. The simplest MLP has three layers: an input layer, a hidden layer and an output layer. The input layer's activations feed into the hidden layer which in turn feeds into the output layer. The output layer outputs the network's classification. Each layer contains individual nodes such as the ones in the perceptron except they have activation functions other than the simple threshold.

Activation functions are similar to a population response of many threshold neurons in the brain. Activation functions mirror the gradual increase and subsequent saturation of the overall activation as the number of neurons that fire increases. For an artificial neuron, the neuron's summed output, $\boldsymbol{wx} + b$, is passed through the activation function ϕ which performs a non-linear transformation of the output. Example functions include the logistic function, tanh function or ReLU function. In our work, we use the ReLU activation function. The formula is:

$$\phi(x) = max(0, x)$$

MLPs use backpropagation to learn. Backpropagation is a way to assign error to nodes in a neural network derived from the error (known as loss) between the output's activations and the target labels in a neural network. For example, if the

output layer's node is 0.1 and the y value is 0, the error is $(0-0.1)^2 = 0.1$. The error is "propagated" back from the output nodes to previous nodes. The error is used to adjust the weights of the neural network to minimize the error at each node. In this way, the output nodes produce output closer to the target labels after each iteration of backpropagation.

The weights are adjusted using the following formula:

$$\Delta w_{ij} = w_{ij} - \eta o_i \delta_j$$

where η is the learning rate which scales how much the weight is adjusted and δ_j is error of the outgoing node and o_i is the incoming value to the weight.

The delta is calculated as follows if using the logistic function:

$$\delta_i = \begin{cases} (o_i - y_i)o_i(1-o_i), & \text{if output node} \\ \sum_{j \in J} \delta_j w_{ij} \cdot o_i(1-o_i), & \text{if inner node} \end{cases}$$

The $o_i(1-o_i)$ is the derivative of the logistic activation function.

2.1.4 Softmax

Softmax operator is useful to turn the neural network outputs into an equivalent probability distribution where each node's output becomes the probability that the network believes that particular output corresponds to the correct label. The softmax operator transforms the output values to such a distribution. The exact function is as follows:

$$o_j = \frac{e^{o_j}}{\sum_k e^{o_k}}$$

2.1.5 Cross-entropy error

As with most supervised learning algorithms, the goal is to predict $P(\boldsymbol{y}|\boldsymbol{x})$. To do so we use the principle of maximum likelihood estimation. The idea is to find a model distribution that best represents the training data distribution. We wish to maximize the probability that the model fits the training data. For reasons of efficiency, the log likelihood is used instead. To measure the difference between the two distributions we use the negative log likelihood also known as the cross-entropy error.

2.1.6 Training

The network undergoes many iterations of small changes to its weights and biases to find the best network classification accuracy. In essence, the training is an optimization problem where the goal is to minimize the error of the network by changing the weights and biases of the network. The weights and parameters are adjusted based on the feedback from the error of the network on the training data.

The network normally begins with a random assignment of weights and biases. Then a training instance is given to the network and run through the network, the final activations of the output neurons are compared to the expected output of the training instance. The CE error is calculated and then the gradient of the error is subtracted from the weights and biases. This moves the weights and biases values closer to producing the desired output. The loss function is not guaranteed to converge to a global minima but in practice performs reasonably well. We use the adam optimizer which is a first-order gradient descent optimizer [22]. The algorithm adjusts the current gradient step by weighting it by the moving average and variance of past gradients.

2.1.7 Parameter Initialization

The weights and biases at each layer l are usually initialized based on the number of nodes connected to layer l. The idea is to keep the variance of each layer's weights the same to speed up training. This technique is known as Xavier initialization.

$Var(W) = \frac{2}{n_{in}}$, where the variance of the weights initial values is equal to 2 over the number of input nodes.

2.1.8 Validation and Testing

It is important to know the accuracy of the network on unseen data as that is a better measure of how well it will perform in a real world setting and see that the network has not just memorized the data. This data is known as validation or testing data. Validation data is used to help select a model by training the model on training data and then seeing the error rate on validation data. The model is then adjusted to perform better on the validation data. In this way, the validation set "leaks" into the

model building process and biases the model towards favoring models that perform well on the validation data. Therefore, at the end of model building, the model is run on testing data which it has never been exposed to in order to evaluate the true error rate.

2.1.9 Batch Learning

The whole dataset can be used to compute a single gradient descent step, however this is often computationally infeasible. Instead stochastic gradient descent can be used, which uses only a single training example at a time, however it may get stuck in local minima more easily as the computed gradient is an approximation of the true gradient. A compromise between the two is to adjust the weights and biases using the average gradient of n random instances from the training data. Each successive iteration of the network is trained on a different small sample of the data called a mini-batch. We use the mini-batch method.

2.1.10 Learning rate

The learning rate is the magnitude of the gradient that is subtracted from the weights and biases. A large learning rate may learn faster but runs the risk of overshooting the intended direction of the minimization problem. A small learning rate such as 0.001 is generally preferred for convolutional neural networks.

2.2 Convolutional Neural Networks

A convolutional neural network is composed of layers of spatially-invariant feature maps or kernels. These feature maps learn to detect specific features anywhere in the input channels. In this way, small disturbances between training instances are ignored e.g. an eye feature detector is able to cope with the minor variations of the location of eyes on human faces. Successive layers of learned feature detectors are stacked on top of each other so that the higher layers are hierarchical compositions of lower level features. The final layer is a fully-connected layer similar to the hidden layer in a MLP.

Convolutional neural networks (CNN) were first popularized by Le Cun et. al. in

1998 where they showed that they could perform well on the MNIST letter dataset [26]. An earlier version of the convolutional neural network was created by Fukashima called the Neocognitron [13]. CNNs did not become popular until 2012 when several key contributions were made by Krizhevsky and Hinton who harnessed the growing power of GPUs and a technique called dropout to outperform the state of the art on a large image classification task called Imagenet [24]. Since then CNNs have become extremely popular over the past five years and are used in many real world problems such as the aforementioned image classification, speech recognition and machine translation.

2.2.1 Convolution

Each layer is composed of a set of convolutional operators. The convolutions are x by y sized matrices called kernels that pass over the entire image and on each pass the kernel values are multiplied by the input, summed up and added to the bias. The new value is placed in the output matrix.

In order to preserve the size of the original image we add padding around the image. Padding entries are commonly zero valued entries around the matrix.

An example of a single convolution is shown in Figure 2.1. Here a 3 by 3 image is convolved with a 2 by 2 kernel. The image has a 1 pixel wide padding.

A convolutional layer is composed of multiple n kernels that produce m output matrices called channels. The number of pixels that is skipped between each shift of the kernel is known as the stride.

2.2.2 Pooling

A pooling layer is where a pooling kernel is passed over the output of a layer and "pools" the output into a smaller, compact representation of the output. It reduces the number of parameters and adds invariance to local shifts in features. There are several popular types of pooling. Max pooling takes the maximum element in the pooling kernel. Average pooling returns the average value in the kernel. A standard pooling layer is a 2 by 2 kernel with max-pooling which results in halving the input size.

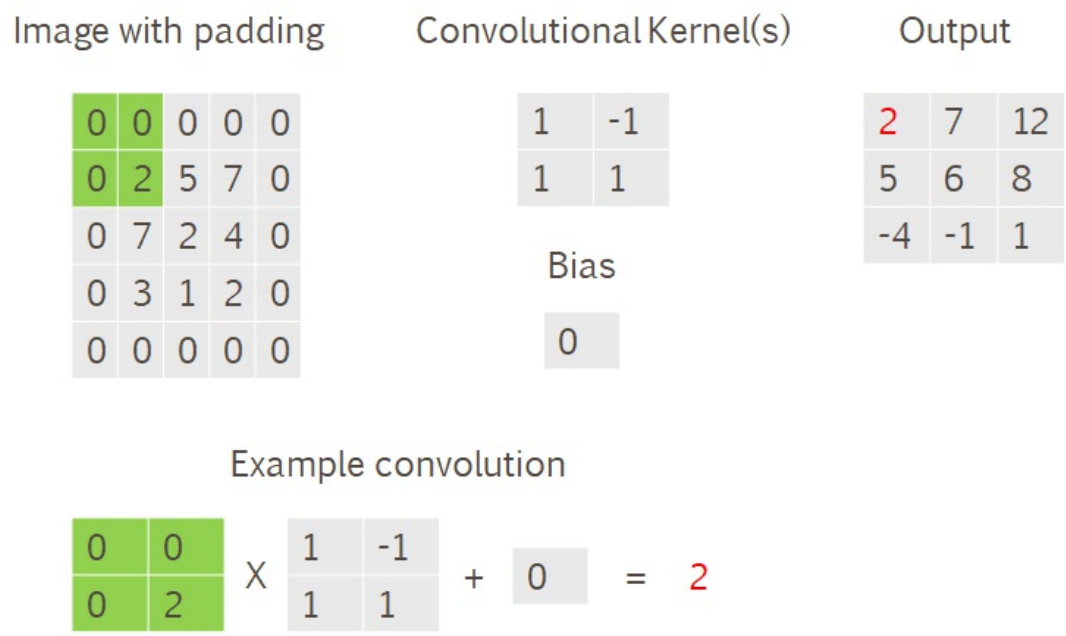

Figure 2.1: Demonstrates an example of a single convolution. A 3 by 3 image is convolved with a 2 by 2 kernel. The image has a 1 pixel wide padding. Only the first convolution is shown. The kernel will pass along the image with a stride of 1 until all rows and columns have been visited.

2.2.3 Dropout

Dropout disables a random subset of neurons in a layer from firing during a training iteration. The outcome of applying dropout is that the network generalizes better to unseen data. The idea behind dropout is that it creates many different samples of different networks on each training iteration so it can be thought of as a form of bagging. Bagging is a term used to describe combining the outputs of many different machine learning models and taking the majority's classification.

2.2.4 Batch Normalization

Batch normalization attempts to normalize each hidden layer similar to how we normalize the input layer. Layer i would prefer if layer $i-1$ activations were normalized, but due to the constant fluctuation of the activations from training, layer $i-1$'s activations are not. This fluctuation is called covariance shift. What batch normalization does on each layer is determine a mean and variance of the batch and normalize that

layer along with using two parameters which are learned by stochastic gradient descent. The end result is that the activations per layer will shift less and make training faster.

Chapter 3

Approach

3.1 Introduction

Convolutional neural networks (CNNs) have traditionally been crafted by hand in an ad-hoc manner. Practitioners usually test different network architectures and their parameters in a non-exhaustive way and rarely report any form of parameter tuning in research papers describing new architectures. They usually only report the best classification accuracy attained on a test set. The lack of parameter tuning is due to the fact that convolutional neural networks can take many hours to fully train – making search algorithms that must evaluate thousands of network architectures prohibitively expensive.

Deep learning practitioners have built a consensus based on performance of a few large-scale high-performing networks as to what the best network should look like. The crafted CNNs tend to be long, deep networks which outperform shorter, wider networks [17]. For many years practitioners have done well with these heuristics.

CNNs have the benefit that they naturally perform well with a variety of architectures and parameters, requiring less fine-tuning and are quite adaptable to different image recognition tasks. So fine-tuning was not a priority to practitioners. However, a network architecture chosen heuristically may perform well on one dataset but may not be the best architecture for another similar dataset. Additionally, one may exist that contains less parameters which can also be beneficial - a smaller network is more compact, faster to train and run during deployment. If a practitioner wishes to achieve the absolute best accuracy than a method to search the architecture space using evolutionary algorithms (EA) may be desirable.

Evolutionary algorithms such as genetic algorithms have long been employed in evolving neural network architectures [34, 27], but their use in CNNs have been less prevalent until recently with the advent of faster GPUs [31]. Their potential lies in their ability to optimize the parameters of the network with architecture search.

We propose a genetic algorithm (GA) to search for an optimal network architecture using hierarchical 2D jagged arrays as the network. The GA evolves a population of CNNs with their loss as the fitness. We evaluate whether the heuristics such as long, thin networks hold when analyzing many different network architectures through our use of a genetic algorithm.

This approach is tested on both the MNIST and CIFAR-10 datasets. The results of the evolved network are competitive with the state-of-the-art. Then we present the types of networks and their parameters which tend to be selected.

We confirm past findings that networks that are deep and narrow are evolved. We find that diverse structures perform similarly and that the mutations that occur most frequently in high-performing networks are high growth mutations such as doubling the size of the network.

We also perform an experiment on a two-class problem to observe the resulting network architectures. We find that the GA creates a hierarchical network that parallelizes the output to the two classes.

The remainder of this chapter is organized as follows: Section 3.2 presents the background on genetic algorithms. Section 3.3 presents the related work. Section 3.4 presents our approach.

3.2 Background

In this section, I give an overview of genetic algorithms and the details specific to our approach.

3.2.1 Genetic algorithms

A genetic algorithm is a method of evolving a group of solutions to solve a problem. GAs are inspired by biological evolution whereby a population of organisms' genotypes have a small chance of mutating each generation and those mutations sometimes produce beneficial functions which make the individuals with the beneficial mutations more likely to survive, produce offspring and live on in successive generations. After many generations, novel and unique adaptations to the environment develop in individuals. The genotype is the DNA inside an individual whereas the phenotype is the individual in the wild.

A GA works similarly to their biological counterpart. An abstract representation of a solution is called a genotype. The genotype is what is mutated and crossed over between individuals in the population. When it comes time to test the genotype's fitness, the genotype is converted to a phenotype to test the solution on the problem. The phenotype is the genotype converted to an actual solution - and no longer an abstract form of the solution. The rules for conversion are fixed and known beforehand.

The algorithm begins with a population of genotypes which are initialized randomly. An GA generation consists of several steps: evaluation, selection, mutation, and cross over.

Evaluation consists of converting the genotypes to phenotypes and evaluating the fitness function for each phenotype on the problem. The fitness function returns a value which measures how well the phenotype solved the problem. We use the loss function of a CNN. Lower loss translates to higher fitness. The fitnesses are assigned to the genotypes.

Selection consists of probabilistically selecting the most fit genotypes to be placed in the new generation's population. Only those genotypes that survive into the new population carry on. Those that were not selected are discarded in the old population. The genotypes with higher fitness have a higher chance of being selected.

Mutation consists of making changes to the genotype with a small probability. The probability remains low, usually 1%. If the mutation rate is set too high then the genetic algorithm devolves into a random walk in the architecture space.

Crossover consists of pairing up two genotypes and switching parts of their genotype. The idea borrows from crossover in genes of cells during sexual reproduction. The motivation behind crossover is that parts of well performing genotypes are exchanged together which preserve the exchanged parts' structure. Crossover probability is usually set around 20-30%.

The four steps are repeated until a stopping criteria is met, such as a finite number of generations or a certain accuracy is attained. After successive generations of evolution, high-fitness solutions tend to survive while weaker solutions do not. A basic diagram of a genetic algorithm is presented in Figure 3.1.

To ensure that the population is always improving, there are various techniques

that can be applied that are not strictly biological, such as elitism. In elitism, a portion of the best evaluated individuals of the population are not modified and pass directly to the next population. In this way, the best solution in the population can never get worse between generations.

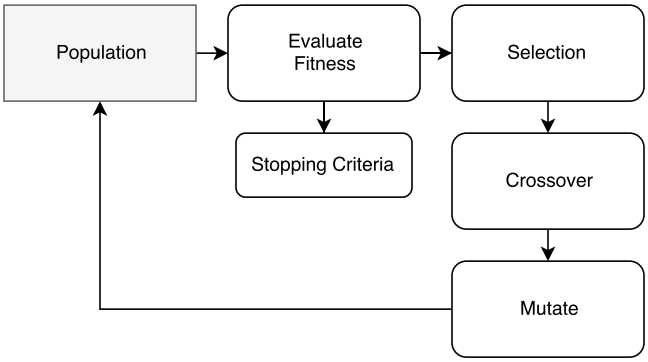

Figure 3.1: Diagram of a genetic algorithm flow.

3.3 Related Work

Parameter search has been used to find the optimal parameters (weights, biases) in a network [7]. The parameter values of a network are modified in an either systematic or random way. Approaches to parameter tuning have been performed such as Bayesian optimization [42] and random search [6]. The end result are highly tuned networks. In our work, architecture search is a form of parameter tuning if you consider the network architecture and number of nodes as parameters of the model.

GAs have long been used to train neural networks [34, 27]. Until recently though, CNNs took too long to train. Now that GPUs have been used to train CNNs, the potential for using evolutionary algorithms like GAs has increased enormously.

We focus the related work on image recognition tasks although GAs have been employed in different domains of Deep learning such as on creating auto-encoders. David and Greental created a deep auto-encoder using genetic algorithms to produce an improved, sparser network [10]. They evolved the weights of the encoding layer. Pinto et al. was on of the first to evaluate many different architectures on image classification tasks for Deep learning [35].

Fernando et al. created PathNet, an example of a modular multi-layered CNN which evolved the connections between modules using a genetic algorithm [12]. They

initialized a set of modules on a finite set of rows and then used a genetic algorithm to select connections between the modules to solve transfer learning tasks. This approach lacks the ability to expand in size or any hierarchical structures contrary to subsequent approaches. One of the goals of this approach is the ability to transfer useful evolved structures to new problems. Task A is learned first and then the weights and biases are frozen and then a new set of connections are initialized while preserving the previous connections. Task B is learned by evolving the connections of PathNet but also using the previous learned connections from Task A. They saw an improvement in training times when training CIFAR-10 and MNIST as a second task. They also applied their approach on Atari games. In comparison to our approach, their GA ran for 500 generations and used tournament selection. Their resulting architecture for the Atari game was 10 modules per layer with 4 layers total. However, the modules were not all connected. The final layer was pre-determined to be a fully connected layer. Their approach was more concerned with transfer learning which differs from our goals of observing the network architectures, so a direct comparison is not possible, but we do find that our GA evolved different sized modules — both in kernel size and channel size and our approach outperformed their CIFAR-10 accuracy. Their results compared to newer approaches shows that perhaps using fixed sized networks is not an ideal technique.

Xie and Yuille created a binary representation of graph structures and then stacked the binary codings on top of each other and called them phases [44]. They used fixed size convolutional filters. They experimented with a 3 phase network on CIFAR-10 and achieved a 76.84% accuracy. Each phase contained around 5-6 modules connected in a graph. They later fit the evolved network structures from CIFAR-10 into larger networks like VGGNet — replacing VGGNet's architecture and significantly improved their accuracy to 92.9%. The downside is that they needed to hardcode the networks themselves and needed a well-designed network to transfer their learned structures. Whereas in our approach we do not need to hard-code the levels or intervene in its creation aside from the initialization phase where we seed the network structures with random modules.

Real et al. designed a GA to evolve network architectures [36]. They do not use any initial conditions and preserve the trained weights of networks through each

generation as long as the weights were not altered during mutation. They utilized a parallel architecture of 250 workers to evolve and train many networks at once. They achieve 94.6% on CIFAR-10. Their mutations allowed for large alterations in the size of the network. Our approach also allows for mutations that add entire layers to the network. They did not use crossover, whereas we did.

Miikkulainen et al. created DeepNEAT, a convolutional version of Co-NEAT, a genetic algorithm for evolving neural networks [31]. The authors evolve two separate groups: modules and blueprints. Modules are small networks and blueprints are graphs where each node represents a module. To assemble a network, they replace each node in a blueprint with the module i.e. a small network. The result is hierarchical graph-like structure. They achieved 92.7% accuracy on the test set. One of the best performing networks featured 12 layers with some linear convolutions (no RELU) and some convolutions plus batch normalization and RELU. They used an initialization method similar to our own where they initialized 25 blueprints and 45 modules before the GA began. Our approach differs in that it uses a hierarchy of jagged arrays rather than a graph so the mutations are different. They also restrict their approach to a certain set of hyper-parameters such 32-256 range of filters and kernel size of 1 to 3.

Simonyan et al. evolved neural network architectures by using a hierarchical encoding of operations in a graph [29]. They reused architectures in different layers of hierarchy which is most similar to our own work, but they use a graph-based structure and use a set number of lower level motifs. They use 200 GPU workers to evolve their architecture. They achieve a 96.37% accuracy on CIFAR-10.

Reinforcement learning has also been used to find the best structure of a network. Baker et al. created a reinforcement learning solution using Q-learning and epsilon greedy search [4]. Actions of the Q-learner consisted of adding a new type of layer to the network. The validation accuracy served as the reward. Over many runs, the Q-learner learned specific constructions of well-performing networks. Zoph and Le used reinforcement learning to search effective architecture space for CNNs [46]. They call their method Neural Architecture Search (NAS). They used a RNN to predict architectures and then use the resulting validation accuracy as the reward for the architecture. The list of generated architectures becomes the list of actions taken

by the reinforcement algorithm. Each subsequent run would learn which pathways were successful and improve the way the network was constructed. Zhong et al. employed a similar Q-learning algorithm [45]. The Q-learning agent sampled different block structures composed of convolutional operators and then trained them in an asynchronous manner. They achieved a 96.4% accuracy rate on CIFAR-10. They used 32 GPUs. Zoph et al. employed a similar idea as Xie and Yuille and first evolved a neural network architecture on CIFAR-10 which achieved a 2.4% error rate (state of the art) and then used the network as a unit itself and stacked multiple networks on top of each other each with their own parameters and tested the stacked network on ImageNet and achieved a state of the art result of 82.7% [47]. They used Neural Architecture Search (NAS) for the evolution of the smaller CIFAR-10 network. They employed a pool of 500 GPUs.

Among the other types of approaches, Assuncao et al. separate evolution into an outer layer consisting of the sequence of layers and an inner layer consisting of the parameters of the layers. They define the allowable range of parameters with a context free grammar. [2]. Liu et al. uses another approach different from genetic algorithms and reinforcement learning. They use sequential model-based optimization (SMBO) to search network architectures of increasing complexity while guiding the search with a heuristic function [28]. Hypernetworks use a smaller network to create the weights of a CNN or recurrent network [16]. Mocanu et al. evolved sparse representations of their Deep learning networks [33] which is in contrast to other work which assumes fully connected layers.

Our work differs from the above works in that we use a jagged array rather than a graph-based structure. Our results do not match the level of accuracy that some of these networks achieve, however our goal is to investigate the resulting architectures of our genetic algorithm to see what patterns arise.

3.4 Approach

In this section, the approach, datasets and preprocessing of the data are presented.

Our GA's task is to evolve a neural network that performs well on an image recognition problem. Therefore, the population of the GA consists of neural network genotypes. A neural network genotype for this problem is an abstract representation

of a neural network. The phenotype is an actual neural network implemented in TensorFlow [1].

The GA population consists of 80 neural network genotypes which are initialized with random structures. Then the algorithm repeats the four step process of evaluation, selection, crossover and mutation until it has reached a limit determined by time allowance and accuracy.

The evaluation consists of converting the genotype into the phenotype, training the phenotype on image classification for a small number of iterations (typically much less than a fully trained network), evaluating the loss and applying the fitness function to the genotype. Selection occurs by probabilistically selecting genotypes that have higher fitness. The probability is calculated by dividing each genotypes' fitness by the total fitness of all genotypes. The more fit genotypes are more likely to be selected. Two genotypes are selected probabilistically and then crossed over, then each are given a chance to mutate. These four steps are repeated until a stopping criteria is met. The training is stopped after 28 generations for MNIST and 101 for CIFAR-10. We use a technique called "early stopping" whereby if the validation accuracy does not change over several generations then the GA is terminated.

After the GA finishes evolving, the neural network phenotype in the population is selected with the highest fitness and then fully trained. Training for 100K iterations on CIFAR-10 and 30K iterations for MNIST. We evaluate them on test data. Once again, we employ early stopping.

In essence, the approach can be thought of as having two distinct learning phases. The first phase of learning is the genetic algorithm as it attempts to find an optimal network architecture and the second phase is taking the best performing network from the GA and fully training it.

3.4.1 Genotype

The genotype is composed of four arrays: the modules array, the structures array, the weights array and the biases array. We refer to the individuals in the arrays as elements. Figure 3.2 provides a visual example of a genotype. The population of genotypes are initialized with random values.

The genotype is hierarchical and modular in nature. The hierarchy is inherent

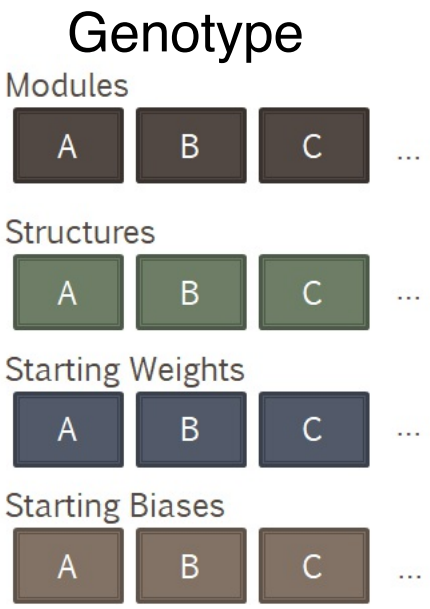

Figure 3.2: Genotype organization: There are four arrays - a structures, modules, weights and biases array.

in the structures. Each structure can contain other structures. The modularity comes from the fact that structures can contain the same modules repeatedly. The modularity allows reuse of the same pieces across multiple locations in the genotype, and allowing a mutation in a modular component to affect all parts of the genotype that uses that modular component. Therefore, one small change can have large consequences in the genotype. The hierarchical nature allows natural structures to form and then be easily swapped to other genotypes in the crossover operation.

Structures

The structures array is composed of structures. It is an expandable n sized array containing any number of structures. A structure is a two-dimensional jagged array. A two-dimensional jagged array is similar to a matrix but whereas a matrix's rows are of equal length a 2-D jagged array's rows can be any length. They can alternatively be thought of as a list of lists. For example the following is a 2-D jagged array with two rows, the first row has 2 elements and the second row has 3:

$$[[0, 1], \\ [1, 1, 2]] \tag{3.1}$$

The rows of the array represent layers of a neural network and can be any length in size. Each row contains any number of modules or structures. For example, if a structure is composed of two rows, the first containing 2 modules and the second array containing 1 module then the structure would represent a 2 layer network with 2 modules on the first layer and 1 module on the second layer. Figure 3.3 demonstrates a possible hierarchy.

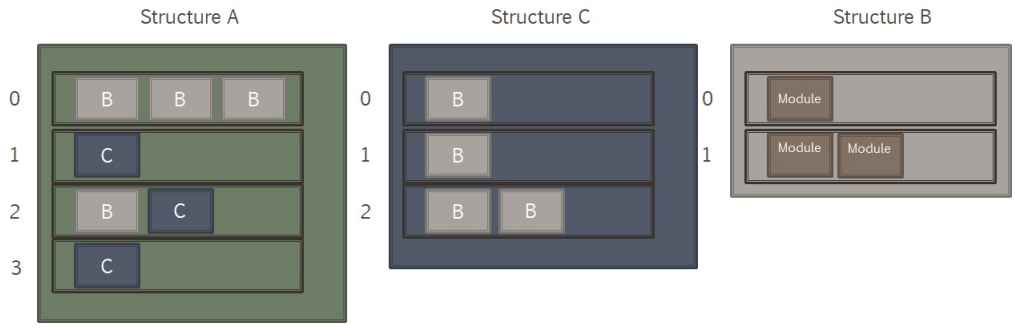

Figure 3.3: A structure with a depth of 3.

Each layer in a structure is fully connected to preceding and succeeding layers. Additionally, skip connections between layers are also permissible.

There is potential for a hierarchy of structures to exist whereby a parent structure contains a child structure on one of its layers. In this case, the parent structure's preceding layer to the child structure is fully connected to the child structure's first layer, then the child structure's output layer is fed to the next layer of the parent's structure. The goal is to allow embedding of deep layers within a single parent layer. The only restriction for structure's children elements is that a child element cannot be the parent element nor can the child element contain the parent element to eliminate the possibility of infinite recursion.

The elements inside a structure's jagged array are merely references to either a module or structure from the modules or structures array respectively. The reference is a pointer to the actual element located in its respective array. That means when a module in the modules array is changed, all the references to that module in other

structures are changed as well. This allows one small change to potentially alter many structures at once. It also allows for a more compact and efficient representation and makes it much easier to add and remove references to structures. To keep the sizes of the structure manageable we restrict the hierarchies to a depth of 4, but our implementation is not limited to any depth and could in theory use a limitless depth.

Modules

The modules array contains modules. There are two types of modules. The first type is called a CNN module and it is a collection of convolutional operators. The full list of parameters for a CNN module are: the number of output channels, kernel size, stride length, 2x2 max pooling (ON or OFF), concatenation of incoming modules (ON or OFF), the weight element and bias element. The second type is called a fully connected (FC) module and is a set of fully connected nodes. A FC module has the number of nodes and a weight and bias element as parameters.

A module is synonymous with a standard layer from a CNN or neural network. For example, a FC module could be a 512 node fully connected layer or a CNN module could be a 64 channel CNN layer with 5 by 5 kernel. The difference in our approach to a standard convolutional layer is that modules can be stacked together on the same layer of a structure. This allows multiple modules (which create different sized output channels) to exist on the same layer.

Biases and Weights

The biases and weights arrays are composed of bias and weight elements respectively. A bias or weight element contains a value and these elements are attached to modules to modulate the starting value of their biases and weights before training. Before the weight and bias elements modify the starting values, the weights are initialized by Xavier initialization so the weights' values are determined by the number of nodes on the ith and jth layers for w_{ij}. After Xavier initialization is performed, the weight values are adjusted by whatever the value of the weight module is.

3.4.2 Initialization of genotype population

The population of genotypes is initialized with random modules, structures, weight and bias elements. The reasoning behind seeding the genotypes with random elements is that without them the GA would waste time increasing the modules and structures arrays before they could be mutated to something useful.

The modules array is initialized with five fully connected modules of 128, 256, 512, 1024 and 2048 nodes and 360 convolutional modules where each module has a slightly different number of output channels, pooling, stride, kernel size etc. Each module receives a random starting weight and bias element.

The structure array is initialized with 60 random structures. Forty structures consisted of a random number of layers with one module each while twenty were two layers long with a one or two modules per layer.

The biases and weights arrays are initialized with 50 starting bias and weight elements. They are randomly assigned starting values with a normal distribution.

3.4.3 Phenotype

The phenotype is a TensorFlow graph constructed from the genotype's abstract representation of a neural network. Careful consideration was taken to build the graph respecting the modular and hierarchical nature of the genotype. Figure 3.4 demonstrates an overview of the conversion process. To begin the process the *main structure* is converted to a TensorFlow graph. While a genotype's structure array can contain many structures, the first structure in the array is always selected as the *main structure*. Of course, the *main structure*'s layers can contain other structures from the structures array. The GA will always be trying to maximize the fitness of the first structure in the array but the GA will be able to develop structures on the side which may get mutated into the *main structure* later on. If the *main structure* is empty, then the fitness is zero.

3.4.4 Convert genotype to phenotype

The genotype's *main structure* is converted to a phenotype. The process involves turning the abstract representation of a CNN into a TensorFlow graph that can be

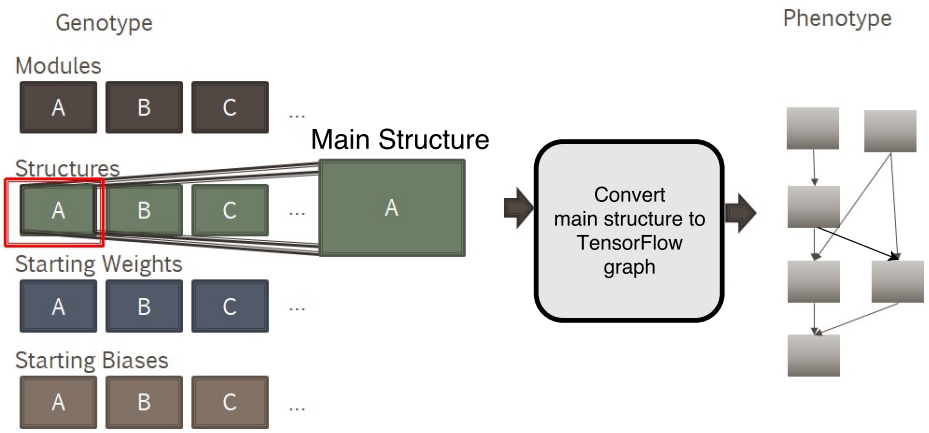

Figure 3.4: The *main structure* (always the first structure in the array) is converted to a TensorFlow graph. The first structure in the structures array becomes the main structure. Then we form a computational graph in TensorFlow by connecting all the modules inside the *main structure*. The process precedes in a feed-forward manner starting from the first layer of the *main structure*. If a child structure is encountered, the child structure's modules are connected recursively to the *main structure's* modules.

trained. It is done in three stages: processing the input layer, then processing the middle layers and processing the output layer. Once the graph is defined, in a separate step, the graph is compiled and ran.

First, the structure's input modules are identified, they are any modules that do not have incoming connections. Each input module is then attached with the input of the problem as Algorithm 1, located in the Appendix, demonstrates. A pointer to the resulting output (graph node), h, is kept in order to store it in the module. In this generic system, a module can have multiple image outputs and store them within the pointer. The usefulness of the pointer is that a module on the next layer can separately convolve over each output image stored in a module's pointer at the previous layer. That way, the modules on a layer are not forced to pool and concatenate together.

After the input has been connected to the input modules of the *main structure*, the rest of the *main structure* is connected to the TensorFlow graph. The graph is created by visiting each element in the *main structure* starting at the *main structure*'s first layer. If a child structure is encountered it is expanded and explored recursively. Each time a module is visited, the matching TensorFlow object is created and attached to

the TensorFlow graph. The module is connected to the modules in previous layers. Algorithm 3, found in the Appendix, shows the pseudo-code for connecting modules with previous modules. The genotype conversion is complete once all elements in the *main structure* have been visited (including child elements). The search through the *main structure* proceeds in a feed-forward manner. Algorithm 2, located in the Appendix, demonstrates the pseudo-code for recursively connecting every module in a structure.

For the final stage, the output modules of the *main structure* are connected to the class nodes (each node representing a class of the problem. A global pooling operation pools all the values from the output modules to each class node — which acts as the final softmax layer. Algorithm 5, located in the Appendix, demonstrates connecting the output modules to the softmax output.

The resulting TensorFlow graph is now connected to the input layer and class layer for the given problem. For image recognition tasks, the input layer is the size of the images and the class layer's size equals the label size. The input and class layer do not change during any phase of learning. For example, in the CIFAR-10 dataset the input layer's size is 28 x 28 x 3 and the class layer's size is 10.

Concatenation

Normally, input to convolutional layers are required to be the same size. In our CNN architectures, multiple modules on a single layer are allowed and these modules may produce different sized image channels e.g. module a has a stride length of 2 so produces a 5 by 5 image while module b (also on the same layer) has a stride length of 1 and produces a 10 by 10 image. We devise two solutions to allow convolutions over different sized inputs, the two solutions are demonstrated in Figure 3.5.

The first solution is for the module on layer $n+1$ to convolve over each separately – creating two output images. However, if there were two modules at $n+1$ they would both now have two different image sizes and that means the module at $n+2$ will then produce four output images. The number of parameters quickly becomes intractable to handle.

The second solution is to downsample or pool the modules of a layer then concatenate the channels - at TensorFlow conversion time - so that the layer contains a

set of same-sized output channels. Then the next layer's modules output channels is not doubled in size. Algorithm 4 shows the pseudo-code for this solution.

Both solutions are allowed and we include a concatenation parameter on every CNN module to control whether it is on or off.

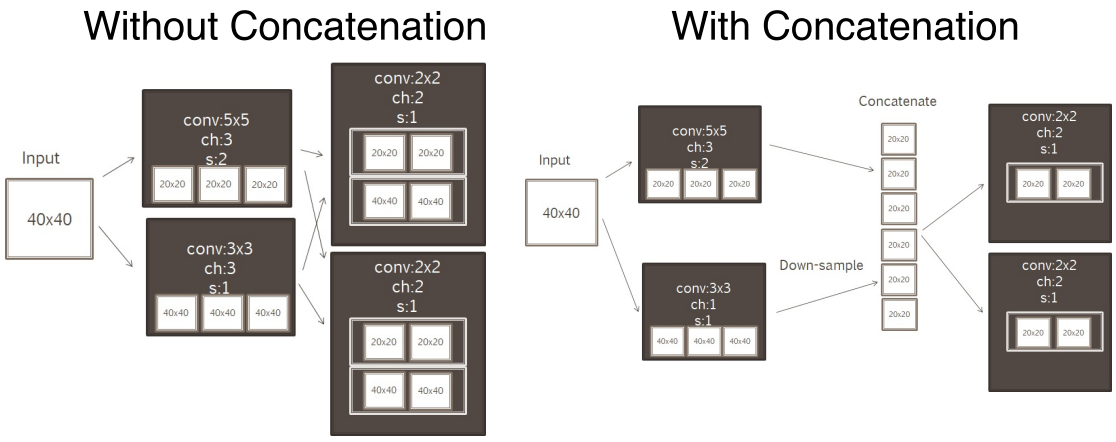

Figure 3.5: Demonstrates two different ways to solve the problem of having different sized outputs on modules of the same layer. In the example, the two modules at layer 1 have different sized outputs (20 and 40). The first solution is to store the results from convolution separately at each module on layer 2. The second solution is to downsample the modules to 20 by 20 and then concatenate the channels together. This allows the second layer modules to convolve over them.

3.4.5 Skip connections

Skip connections are connections which skip at least one layer. They can be added to structures via mutation. Skip connections that were added in the genotype are also added in the phenotype. They provide a means to connect non-adjacent layers. They have been shown to be beneficial in other networks [19].

3.4.6 Evaluation

The converted TensorFlow neural network is then trained on an image recognition task. We use a small number of iterations to keep the overall cost of evaluation down. We begin training for 1800 training iterations using stochastic gradient descent.

If a network is not feasible it is assigned a fitness of zero. Each genotype must have at least one module in its *main structure*.

Once the network has been trained, the loss of the network is used as a parameter for the fitness function. The output layer uses a cross-entropy loss function which is a measure of error between the true labels and the output of the network. The error is backpropagated through the network's weights and biases.

The fitness function minimizes the loss of the network but also the size of the network parametrized by the number of generations. The total parameters are calculated by summing up the number of weights and biases that the resulting TensorFlow network contains.

The size of the network is compared against a threshold value $size_{min}$. If the network's parameter size is less than $size_{min}$ than the fitness is not punished and it equals 1/loss. If the fitness is between $size_{min}$ and $size_{max}$ then a *punish* value is normalized out of 1 between the two threshold values and the *punish* value multiplied by the fitness. If it exceeds $size_{max}$ then the fitness is zero. The $size_{min}$ is 2 million and the $size_{max}$ is 55 million allowing for very gradual regularization. Punishing the size of a network acts as a regularization measure to pressure the GA to evolve smaller networks first and only expand as more parameters are needed.

There is also a *max threshold* where a network cannot exceed; if it does the network is assigned a fitness of zero to stop extremely large networks from running. The value is 5 million and 32 million for MNIST and CIFAR-10 respectively.

3.4.7 Training over Generations

After each generation, the $size_{min}$ (if the size of the network is over $size_{min}$, the network's fitness is reduced) and $size_{max}$ (after $size_{max}$ the network's fitness is 0) are increased by 200,000.

Additionally, we increase the number of training iterations each generation by 40. This way the solutions are trained progressively longer and give better estimates of their true fitness.

After training is complete, the loss, accuracy and fitness (based on the loss and network size) are calculated, then the weights and biases are reset. So the network - if it is reused again in the GA - must learn from scratch before re-calculating its fitness.

3.4.8 Selection

We employ fitness proportionate selection for creating the new population from the old. The probability of selection is:

$$p_i = \frac{f_i}{\Sigma_{j=1}^{N} f_j}$$

where f_i is the fitness of an individual and p_i is the probability of being selected.

Then they are sorted by their proportional fitness and a random number is drawn where its value corresponds to a particular individual.

The GA uses *elitism* to ensure the overall population fitness doesn't get worse between generations and that the population doesn't lose potentially strong individuals at random. The best 10% of genotypes are carried over between generations.

Selection is performed twice, and the two selected genotypes are crossed-over, mutated and placed in the next population. This process is repeated until the new population is filled.

3.4.9 Crossover

We perform crossover on two genotypes at a time. The elements in the element arrays (structures, modules, weights and biases) are aligned starting at the 0th position and swap with the other genotype with a 20% chance. Crossover stops once the end of an element array is reached.

Table 3.1: Mutations for the structures array

Structure Array Mutations	
Create	new structure, a new copied structure
Merge	two structures side by side, two structures vertically, two modules side by side, two modules vertically
Swap	positions of structure, starting position in array
Shift	structure left in array, structure right in array
Cut and rearrange	a split of the array
Remove	structure
Transfer	structure references to another structure (keep old), structure references to another structure (remove old)

Table 3.2: Individual weight and weight array mutations

Weight Mutations	
Adjust	individual value
Double	individual value
Divide	individual value
Create	new weight, a new copy of a weight
Transfer	references to another weight (keep old), references to another weight (remove old)
Remove	weight from array

Structures

Due to the unique hierarchical and modular nature of a structure, we employ a novel method of crossover. Elements that are swapped must remove their child references to their old genotype and replace the lost child references with new references from the new genotype (at the same location as the old references). For example if a structure s was swapped to genotype b from genotype a, then it removes its child modules from genotype a and replaces them with corresponding modules in the same location from genotype b. The replacement modules should be in the same location as the replaced modules from the old genotype. If not, they are removed. Figure 3.6 demonstrates a simple example where a a structure is swapped and then its child element is swapped with the respective child element in the new genotype. Algorithm 6 demonstrates the pseudo-code for crossing over structures.

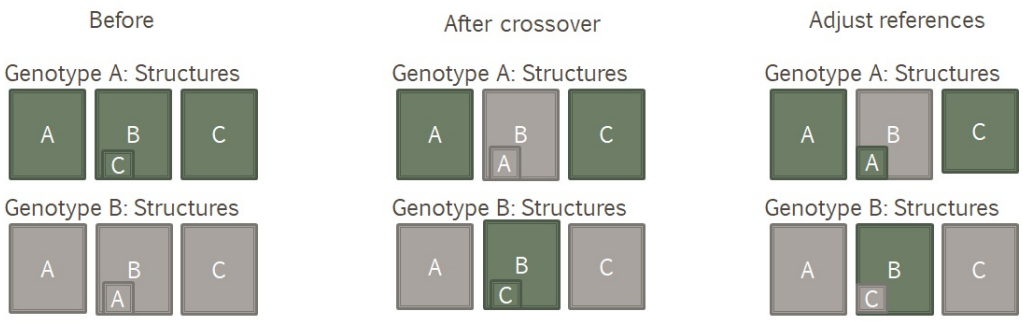

Figure 3.6: Demonstrates two genotypes's structures undergoing crossover. Here one structure is swapped with the other structure at the same position. The child element of each structure needs to be swapped as well because they reference structures from their old genotypes.

Table 3.3: Mutations for a structure's jagged array

Structure's Jagged Array Mutations	
Insert	module as a new layer (first), module as a new layer (last), a module as a new layer (anywhere), module on exiting layer, structure as a new layer (first), structure as a new layer (last), a structure as a new layer (anywhere), structure on exiting layer
Create	copy of element in layer and insert
Double	size horizontally, size vertically
Copy	layer elements then append to the same layer, layer elements then add as a new layer (below), layer elements and add as a new layer (above)
Overwrite	a layer with a random module, a layer with a layer module or structure, a column with a random module, a column with a random structure or module, all elements with a module, all elements with a structure or module, a layer with a weight, a layer with a bias, a random element with a module, a random element with a structure, biases of elements (non-recursive), weights of elements (non-recursive), a layer with a random structure, a column with a random structure, all with structure, a layer with a single random module, a layer with a single random structure
Remove	layer, module or structure from random location, all elements, all but one element from a layer
Set	a module's pooling to true, a module's pooling to false, a module's concatenation parameter to true, a module's concatenation parameter to false
Add	random identity skip connections

Modules

When two modules are swapped between genotypes, structures that reference them must reference the other swapped module. For example, if module a is swapped with module b, a structure that referenced a will now reference b. The biases and weight elements in a module a must now refer to biases and weights in the new genotype (if possible) - otherwise they use a default value of 0. Algorithm 7 demonstrates the pseudo-code for crossing over modules.

Weights and Biases

When weight or bias elements are swapped, any modules referencing them must reference the new swapped element.

3.4.10 Mutation

During the mutation phase, each element in the genotype undergoes mutation. The mutation operators have a 1% chance of occurring. The mutation operators modify the location of the elements in the four arrays and the parameters of the elements themselves. We have included the names of each mutation operator in Tables 3.1, 3.2, 3.3, 3.4, 3.5 for reference. The bias mutation table is identical to the weight mutations so we have not included here. For the full list of mutations and more detailed explanations see Appendix B.

Table 3.4: Individual module mutations

Module mutation	
Change	a module's weight initialization, a module's bias initialization
Add	1 node to fully connected module, 1 output channel to CNN, 1 to kernel size, 1 stride in CNN module
Subtract	1 node of a fully connected module, 1 output channel to CNN, 1 to kernel size, 1 stride in CNN module
Double	number of nodes, output channels
Halve	number of nodes, output channels
Flip	pooling, concatenation

Table 3.5: Modules array mutations

Modules Array Mutations	
Create	CNN module, FC Module, copy from existing module
Transfer	references from one to another (keep old), references from one to another (remove old)
Swap	two modules in the array
Shift	modules left, modules right in array
Remove	module from array

3.4.11 Data Selection

We evaluate our approach on both the MNIST and CIFAR-10 datasets [25, 23]. We select them because they are popular benchmark datasets for Deep Learning.

The MNIST dataset contains 50,000 28 by 28 pixel images and 10,000 test images. Of the training data, 45,000 was used for training the networks and 5,000 for validation

of the GA. The test data was set aside for evaluating the approach so as not to bias the results. There are 10 classes for each of the 10 digits. The images are 1 dimensional gray scale images. We used a batch size of 50. All images are randomly cropped to 27 by 27 and normalized which means that the image's mean pixel value is subtracted from each pixel value then divided by the standard deviation of all pixel values in the image.

The CIFAR-10 dataset contains 50,000 images 32 by 32 pixel images and 10,000 test images. There are 10 classes: motorcycle, car, etc. The images are 3 dimensional color images. We use 42,500 for training, 7,500 for validation and 10,000 for testing. We use a batch size of 128. The images are cropped to 24 by 24 pixels from 32 by 32 and normalized. The images are randomly flipped left or right.

An iteration of training consists of one run of the network on a training batch. Once the network has seen every batch of the training data, the data is reshuffled and preprocessed again. We use batch normalization and dropout.

Chapter 4

Results

In this section we present the results from evaluating the performance of our approach and our observations on the types of structures that evolved.

4.0.1 GA performance

We demonstrate that the GA can evolve high performing networks on different sets of image recognition tasks.

The best performing network achieved an accuracy of 99.5% on MNIST

After 28 generations, the GA produced a network that reached 99% accuracy. Figure 4.1 shows the best network's accuracy each generation. After only the first generation, the network was already at approximately 98%. As Figure 4.2 shows, the mean population also increased over time, but saw more fluctuations then the best performing network. The fluctuations are due to mutations modifying networks and reducing their fitness or increasing their fitness. The best performing network (lowest loss) was selected and fully trained for 30,000 iterations and reached a test accuracy of 99.5%. The accuracy is comparable to the state of the art on the MNIST dataset which is in the range of 99.6-99.8% [5].

The GA evolved a three layer network. The first layer contained 1 module of 511 channels, 3 by 3 kernel size and stride of 1 with 2 by 2 pooling. The second and third layers were identical – two modules of 256 channels each of 3 by 3 kernel size, stride of 2, no concatenation and no pooling. The network had 4,884,480 parameters. Figure 4.3 demonstrates the architecture of the network.

The best performing network achieved an accuracy of 89% on CIFAR-10

After 101 GA generations - which corresponded to 3 week of training - the network with the highest fitness achieved an accuracy of 85%. The average accuracy of the

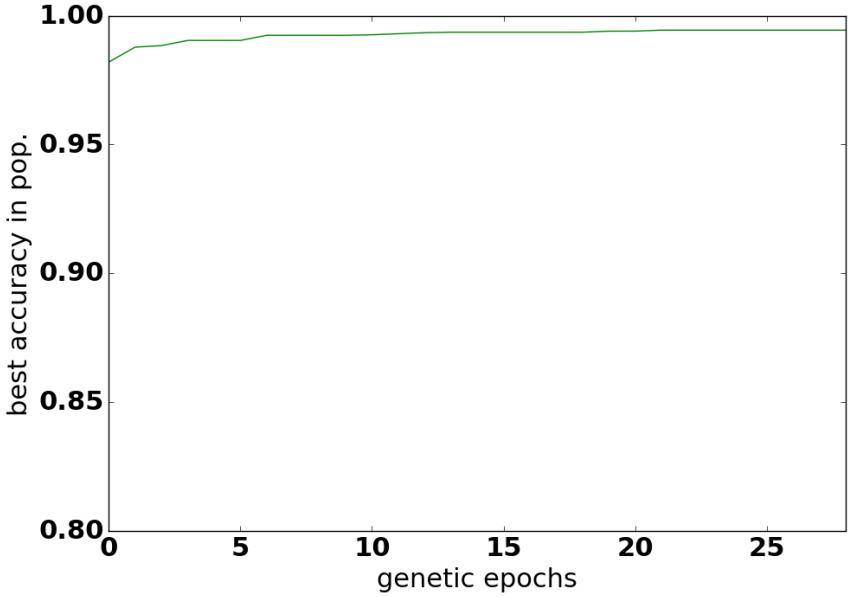

Figure 4.1: Best network's accuracy in the population each generation on MNIST data. Accuracy out of 1.

population was 69% indicating that some networks still performed poorly within the population after 101 generations. Figure 4.4 shows the mean accuracy of the population improved over successive generations. After fully training the best performing network, the network achieves an accuracy of 89% on the test set. Figure 4.5 shows the learning curve of a full training run on the best network. This result remains competitive with the state-of-the-art, but falls short of matching the latest evolutionary algorithms results of 92%-94% accuracy [31, 29]. However, we are more focused on what types of networks are evolved and whether the heuristics of previous networks hold true. So we are not concerned with achieving the best result.

The GA evolves an 8 layer network. Seven of the layers are identical, each composed of two identical modules. The module has 127 channels, 6 by 6 kernel, stride of 1, no pooling and concatenation. The third layer contains 4 modules of 2 types. The first two are the same module type mentioned before, the second has 15 channels, a kernel of 7, stride 2, no pooling and concatenation. Figure 4.6 shows a diagram of the network. Additionally, there is a skip connection from the third to 6th layer.

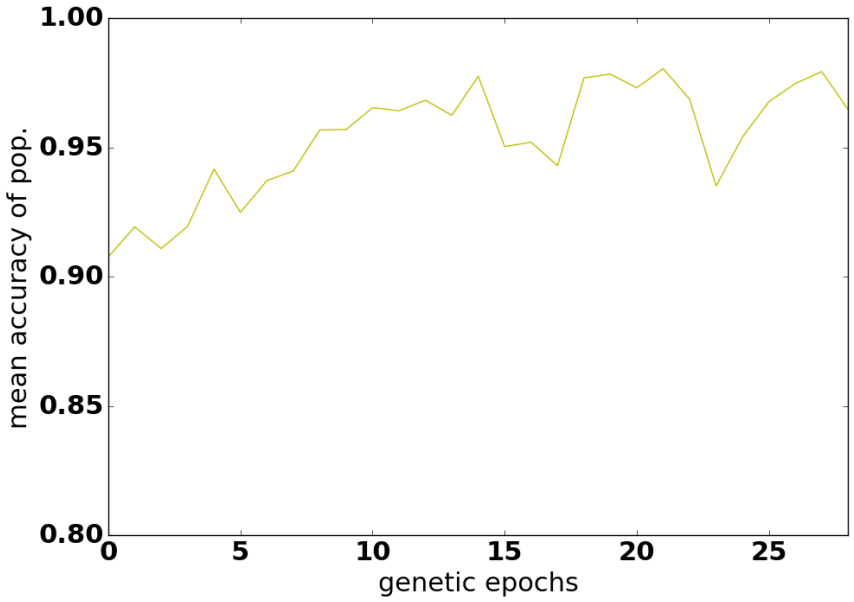

Figure 4.2: The mean population accuracy over GA generations for MNIST data. Accuracy out of 1.

4.0.2 Structure evolution of MNIST

We provide a summary of our observations on the evolved structures for the MNIST dataset. After 28 generations, there is 72 feasible networks of 80. The average parameter size of a network is 2,030,000.

The MNIST networks are short, but have multiple modules per layer

The average number of layers is 3, the average number of modules per layer is 1.7. This means the GA favors width to length. The features of the first few layers of a convolutional neural network are edge detectors and simple features. They lack the abstract representation of the higher layers. We hypothesis that the fewer layers are due to the simpler nature of the MNIST dataset, which is a series of black and white characters.

One hypothesis as to why the network has multiple modules is that it is an effective means to quickly double the number of features by a single mutation. Indeed, we observe that in the best network, its second and third layers do not concatenate, so the addition of 2 modules per layer effectively acts a way to quickly double the

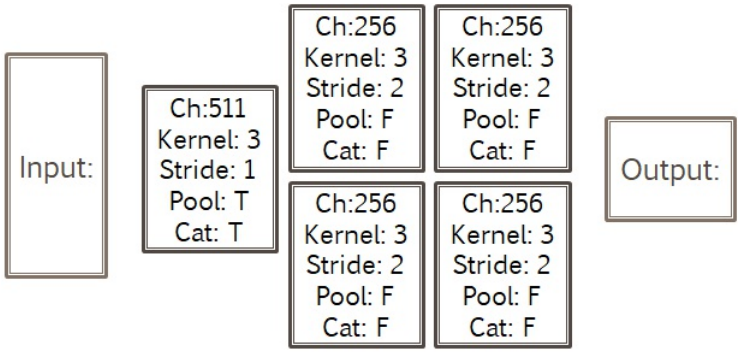

Figure 4.3: Network architecture of the network that achieved the lowest loss in the GA.

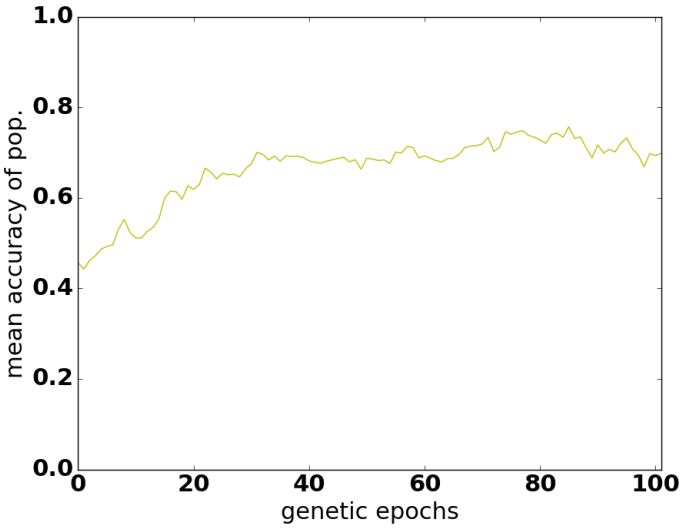

Figure 4.4: The mean population accuracy over genetic generations for CIFAR10 data. Accuracy out of 1.

number of features.

MNIST networks reuse modules

The unique modules per network is 2.4 and the total number is 5.6 which means the networks usually contain duplicate modules in the network. The network reuses the same modules as our approach was designed to do. Many mutations support this type of operation where mutations are copied across a layer. The benefit of reusing a module is that a mutation can change the referenced module which updates all the modules in the network at once, which allows for greater ability for the network to

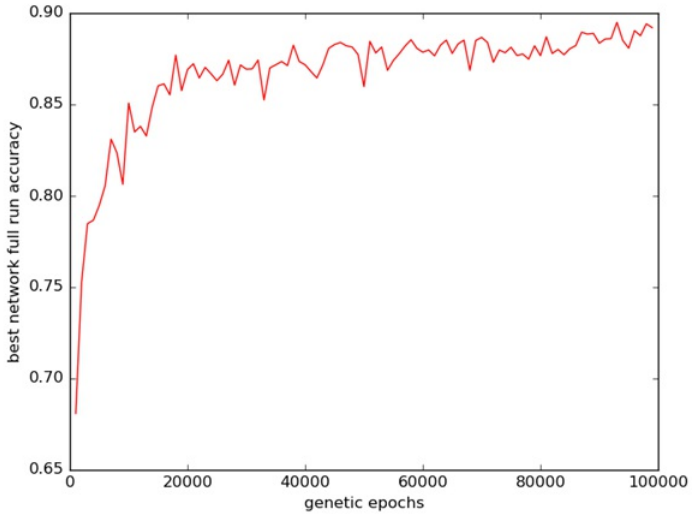

Figure 4.5: Fully training the best performing network from the GA on CIFAR-10 data. Accuracy out of 1.

adjust to the problem.

The evolved structures are shallow

The average depth of the *main structure* is close to 0 (0.17). Hierarchical structures were not as good as shallow structures. We see this pattern repeat among the related works as well. One of the original benefits of hierarchical structures is the powerful mutations that can move entire structures in or out of other structures. But from our experience, they are not used. We hypothesis that the mutations that modify layers are already powerful enough operators.

Population as a whole evolved similarly high accuracy networks

The average accuracy was 96%. This finding highlights how powerful CNNs and backpropagation are and why there was not as much research into architecture search until GPUs were capable of supporting them. A practitioner can devise a network that performs well with a high degree of probability just by using a few layers and a number of feature channels per layer. However, we do observe a range of performance across the networks indicating that there are some networks that perform worse. As demonstrated in Figure 4.7 the number of layers seems to have an effect on the

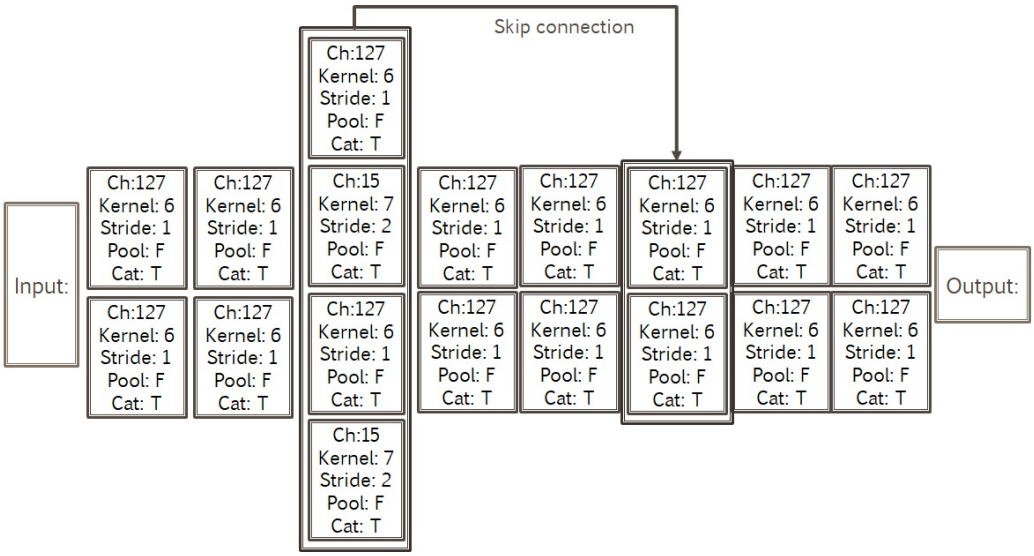

Figure 4.6: Diagram of the best performing network on CIFAR-10 data.

accuracy with networks of length 3 and 4 performing best.

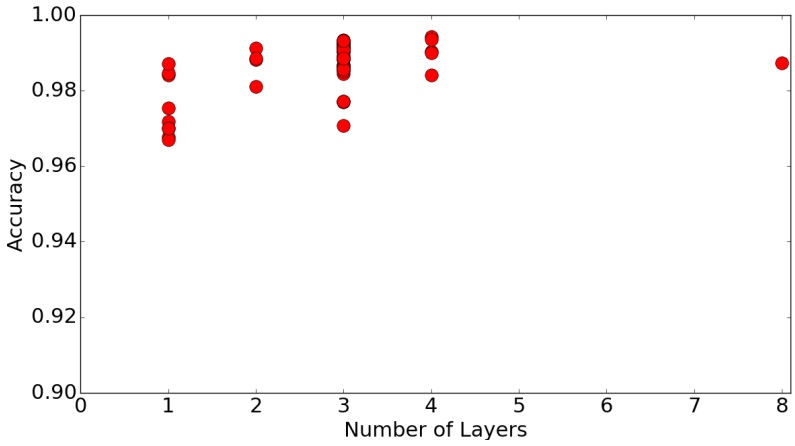

Figure 4.7: Number of layers against the accuracy of networks in the population on the MNIST dataset.

The GA creates networks of modules that conform to common values

The average kernel, stride and channel size is 4, 1.7 and 153 respectively and the standard deviation is 1, 0.4, and 83 respectively. The ranges fall within popular choices for networks. The networks used in the past are in the range of 3-5 kernel size, 1-2 stride and 128-512 channel size. This demonstrates that there may be value

in constraining the parameters during evolution to conform to specific ranges. It would speed up training and not sacrifice performance. Indeed, many of the related work papers included constraints.

The best network's parameters varied across epochs

As Figure 4.8 shows that even as the accuracy increases for the best network, its parameters vary considerably. This shows that the largest networks are not necessarily the best and finding the right fit for the problem is paramount.

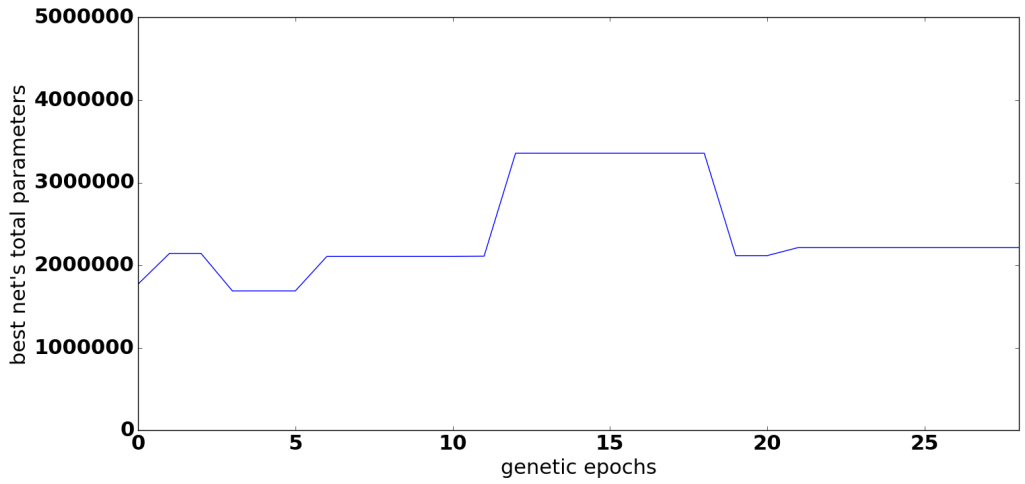

Figure 4.8: Number of parameters of the best network over generations of the MNIST data.

The evolved networks use both pooling and concatenation

55% of the time modules are set to use pooling. As we see from Figure 4.3, the first layer uses pooling but the subsequent layers do not. Pooling helps reduce the size of the network but loses some detail. We hypothesis that for the MNIST dataset a balance of some pooling was required, but too much would lead to too much loss of detail.

32% of the time a module is set to use concatenation. It seems that with the smaller networks created for MNIST, that concatenation was not required as much. For example, the best network's third layer has double the normal weights because two sets of weights project to the second layer's two modules but the network has

only 3 layers in total so the size is not too great (over two million). The size is still large compared to a similar three layer network which would be near half the size in total parameters.

Weights and bias elements go mostly unused

Initialization of the weight and bias elements are only used in 21% and 44% of the modules. The values are -0.8 and -0.89. The low usage rates show a downward pressure from the GA to not use the weight and bias elements — otherwise the rates would be around 50%. We hypothesize that the Xavier initialization is already adequate and so the initialization parameters didn't contribute much to faster learning, so the networks did not evolve to use them.

Fully connected layers are not used

Fully connected layers — while used extensively in deep learning modules — were not evolved in the MNIST dataset. Usually they appear at the last layer, the convolutions are seen as building a representation of the data and then the final fully connected layer is essentially the hidden layer of an MLP where the input data is the well represented features of the CNN. We hypothesis that there were more CNNs in the modules array so it was less likely that a fully connected layer would be mutated into the last layer of the network. In the future, we may wish to experiment with adding an additional mutation to add a fully connected module to the last layer. The other reason may be that the global pooling layer at the end acts as a fully connected layer in itself and so the fully connected layer is not required.

4.0.3 Structure evolution on CIFAR-10

We focus next on our observations of the CIFAR-10 dataset network structures. The CIFAR-10 dataset is far more complex and difficult to classify than the MNIST dataset, so we see a more complex network evolve and it requires more generations to learn.

Parameter size was not always a good indicator of performance

Figure 4.9 plots the best network's total parameters each generation. The best network's accuracy is monotonically increasing each generation, thus if there is a drop in parameter size, we know that the GA evolved a better, smaller network. The parameter size ranges from 30 million to approximately 11 million.

We observe that there is a large flux in total parameters of the best network as it evolves. We observe that similar network accuracies exist between networks of disparate sizes. In several instances, large drops in parameter size results in a higher accuracy. Therefore, we hypothesis that using the largest network possible is not always the best solution. Careful consideration of the architecture can lead to improvement of the network's accuracy. Larger architectures are also slower to train and larger to store in memory, so smaller architectures are more desirable if they equal the larger architectures in accuracy.

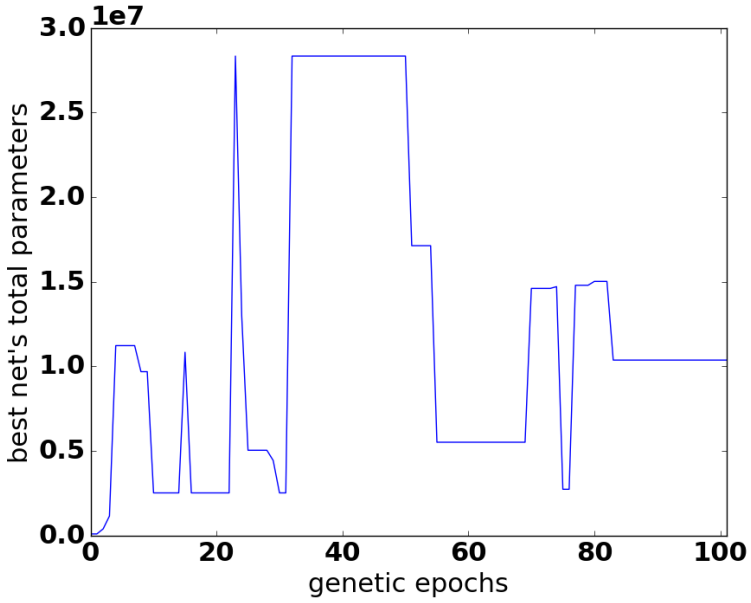

Figure 4.9: The average size of the parameters of the network over each generation of the genetic algorithm on CIFAR-10 data.

The GA produced a diverse range of network structure sizes

The average size of feasible networks was 11.6 million. 11 of 80 networks were not feasible – either because they were over the limit of 32 million or the *main structure*

was empty. The distribution of sizes ranged from tens of thousands to 30 million parameters. We observe that the population of the GA did not converge to a similar architecture. We hypothesis that because 1) the mutations allow for large changes to be affected on the network at once through replacing layers or doubling or halving the size and 2) that a mutation to a module will affect all of its references in other structures ensures that the networks remained diverse.

As can be seen from Figure 4.10, the accuracies see a slight improvement as the size increases to about 15 to 20 million parameters but smaller networks show remarkable resiliency even at much lower parameter sizes - some as low as half the largest network. Again reinforcing the concept that CNNs are capable of adapting to the problem at hand given a less than ideal architecture. However, to achieve peak accuracy there are certain network architectures that are favored.

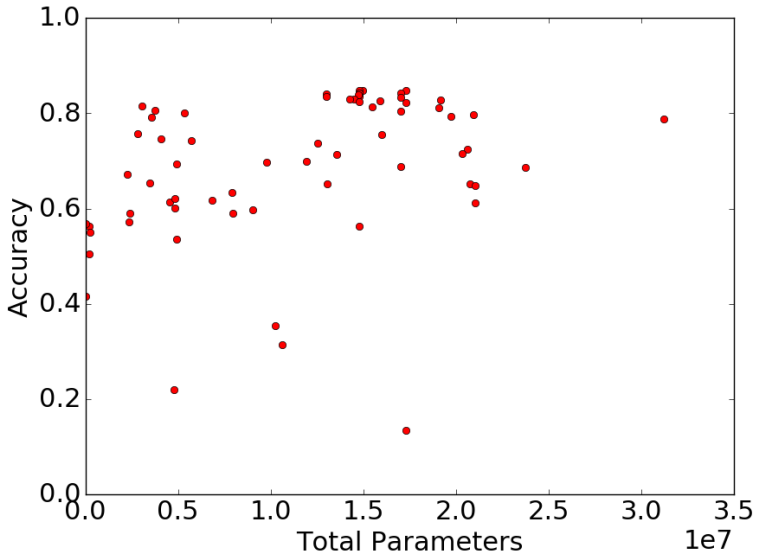

Figure 4.10: Plot of the total parameters of the evolved networks in the population and their accuracy on CIFAR-10. Accuracy out of 1. There is a slight trend towards network sizes of 15-20 million performing better however, the smaller networks still perform reasonably well even with half as many parameters.

Longer networks of 7 to 8 performed the best

The average length of the evolved networks was 7. Additionally, the accuracy of the networks increased as the network's layers increased up until 8 layers. The network

ceased improving past 8 layers. Networks with less layers performed worse, which is in contrast to the MNIST dataset problem which required 3-4 layers to perform the best. The longer networks allow for more hierarchical features to form as the class values are more complex. For example, the objects include trucks and cars which have many discernible parts, textures and colors. Similar lengths were produced in other evolved networks [31]. We cannot say definitely that this is the ideal size for the problem, only that that is what we observed that the GA produced. However, a practitioner can have far more confidence that they are closer to an ideal structure by using the GA than without using one and creating the network by hand.

Multiple modules per layer were present in high accuracy models

The average number of modules per layer was 2 meaning most layers contained only 2 modules. Multiple modules per layer allows for different output sizes if the modules have different stride lengths or opposite pooling values. The modular connections are similar to the Inception network [43] and shows how different modules can be beneficial to present different scales of input data. For example, in the best performing network, we observe multiple modules at layer 3, two have a stride of 2 and the other two have a stride of 1 which produces two different sized images.

The GA favors shallow hierarchies

At least during the generations observed, the GA produces an average depth of 0 – meaning no hierarchies. This shows that hierarchical networks are not necessary for types of image recognition tasks like CIFAR-10. We hypothesis that the complicated structures that a hierarchy could produce simply were not required. It remains to be seen if further generations would produce hierarchical networks.

Structures that exhibit reuse are preferred by the GA in *main structures*

The average number of unique elements (modules/structures) in the *main structures* of genotypes is two while the average total number of elements in the *main structure* is 15. The disparity between total elements vs. unique elements means that the same repeated elements are reused within the *main structure*. Repeated structures are a key component in other models such as in Residual Networks [17].

Average kernel size, stride and channel output were 5.6, 1.5 and 113 respectively.

The evolved networks tended to prefer larger convolutions, a stride of 1 and 2 and large numbers of output channels. The variance for kernel size, stride and channel output is 0.9, 0.69 and 53 demonstrating that there is very little variance between the populations parameters. We observed comparatively larger kernel sizes 6-7 than what is normally used in other works which use 1-5 [45].

The networks used concatenation but not much pooling

Networks contained 92% concatenated modules and only 31% pooling. We hypothesis that the concatenation is present to prevent the number of parameters from doubling each layer. It is especially important for longer networks such as on an 8 layer network due to the number of layers. Pooling occurs less because we believe the lack of pooling helps maintain a high level of detail needed to discern complex features in the images.

Networks didn't evolve the use of weights or bias initialization

We found that most networks did not make use of the weight initialization at only 18%, most likely because the Xavier initialization was sufficient. Same for the biases where the biases were only used 64% of the time. As above, we note that this is evidence that there is evolutionary pressure to remove the weights entirely.

We find that only CNN modules were used versus FC modules

Again, FC layers are not used, we hypothesis that it is due to the low probability of a mutation occurring which adds a FC module at the end of the network.

The most used mutations are those that increase size

We record the types of mutations that occurred on the modules and structures of each genotype. We specifically look at the genotypes of *main structures* because they were the structures used to generate the network. We find that the top five mutations that occur in order of magnitude are: 1) double size of network vertically, 2) copy a layer and reinsert as a new layer, 3) overwrite all elements in the structure with a random

element, 4) set layer modules to concatenation = True, 5) append a random module to the array. Intuitively, we can observe the benefits of these mutations. The 1st, 2nd and 5th mutations increase the size of the network. Overwriting the structure ensures uniformity and applying concatenation reduces the number of parameters for multi-module layers. Table 4.1 shows the mutations which occurred the most in the final genome population on the *main structures*. These three mutations lead to the popular heuristics of very long networks, repeated patterns and pooling different sized outputs together.

4.0.4 Evolution in a modular problem

We devised an experiment to classify the "what" and "where" of objects in an image similar to an experiment by Jacob et al. [20]. We investigated what architecture would result when instead of there being only one classification task there were two. We were motivated to devise this task in part because humans solve problems with separate goals all the time such as locating an object and identifying the object's type. Also, we wished to see whether separate pathways would develop or if the network would remain uniform.

The human brain's visual system has examples of both parallel and sequential processing [8]. In this way, specialized structures handle aspects of the input which are then combined in upstream pathways such as when separate parts of the LGN receive the left and right field of view of each eye. The brain also processes input in a sequential matter where the input is passed through multiple layers of processing, producing abstract and more meaningful representations of the low level features [11] For example, visual information is passed from retinal ganglion cells to V1 cortical neurons and then to higher level processing in the V2 and V3. Goodale argued that there existed two distinct pathways the dorsal and ventral streams also known as the "what" and the "where" pathways [14]. Both the parallel and sequential paths are required for efficient and thorough processing of human visual information.

Evolving the structure of a neural network emulates the evolution of the human brain at a much smaller scale where the structure fits the problem at hand. Just as in the human brain, structures specialize and also share information. We test whether our approach is capable of producing similar structures.

The task we choose to test is a segmentation task which requires the network to classify both the *where*, and the *what* of an object. The neural network must predict from a set of shapes which shape is located in the image and also predict where it is in the image.

More specifically, each training image has two sets of labels - a location label and an object label. The neural network is given an image and must correctly predict the correct label for the location and object label. For the location label, the possible values correspond to every combination of x and y coordinate (x,y). For example, given a 10 by 10 training image there are 100 possible label values because there are a 100 possible combinations of the x and y coordinate. The object label consists of the combination of (yellow, red, and blue) and (square, circle and triangle) for a total of 9 objects. To make the task more difficult we added noise. Figure 4.11 shows a shape with noise.

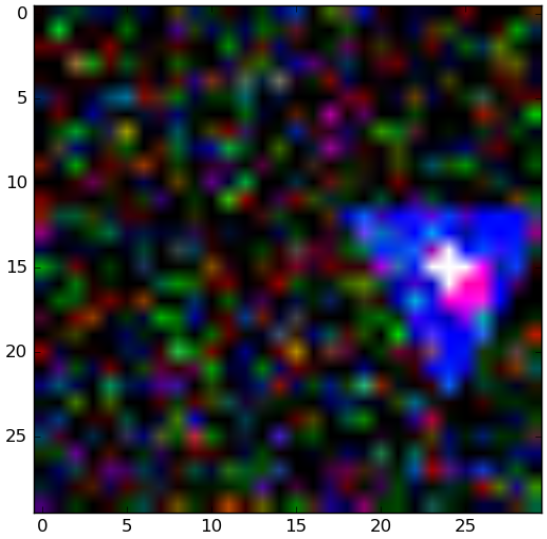

Figure 4.11: Example training image with noise. The training image contains a shape, and the label location that the network is attempting to classify. The white dot doesn't appear in the training image and used only to indicate the center.

We modified the traditional architecture of a neural network to accommodate two classes – the location of the object and the object itself. The label for each image is thus a tuple e.g. ((1,3), 'red square'). The output modules of the evolved structure

are split into two groups — one for each class. So each output module projects to only one class.

To be more specific, the steps are as follows: 1) The output modules are identified. The output modules are any modules that have no outgoing connections to other modules. 2) If there is only one output module it projects to both class outputs. 3) If there is more than one, then the output modules are divided into n groups which equal the number of classes. The division of modules into groups is done in a systematic way so the process is repeatable and consistent across different runs. If there are uneven amount of output modules, the split value is rounded to the nearest integer. For example, if there are five output modules then the first two output modules project to the location class and the last three output modules project to the object class. Figure 4.12 shows an example of the output modules. Algorithm 8 demonstrates the pseudo-code of splitting the output modules into two groups.

The location and object labels always receive independent input from separate output module groups. In this way, the GA is able to evolve either a solution where a single output module projects to both or multiple modules project separately to each class — without favoring either approach. Of course, the internal structures of the *main structure* are also free to evolve parallel or sequential structures.

To calculate the error, both the error of the location and the error from the position are summed together. The errors are not scaled or weighted. Back-propagation is used to learn the weights of the network.

We evolve a population of genotypes using a GA and observe the results. The average size of the parameters is 2,580,000. The average number of layers is 2, however that doesn't explain the whole picture because we find that the average depth of a network is 1. This means that there exists structures within each layer (which contain potentially more modules) which adds to the actual number of sequential layers. For example, a *main structure* could have one layer which contained a structure, and inside that child structure could be a 5 layer network, so the "effective" length of the network is 5. The average depth of 1 for the networks demonstrates that the problem required a hierarchical approach. We hypothesize that depth allows for easier transference of good structures to parts of the structure that require them because a single mutation can move a whole structure to a new location. For this

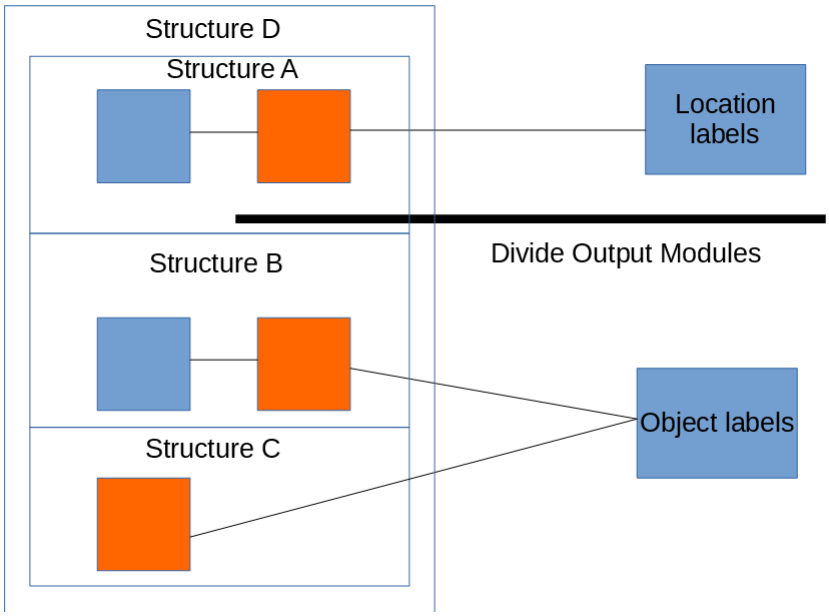

Figure 4.12: The output modules are divided into groups which correspond to the number of label types.

problem with two distinct classes, the hierarchical structures may have made it easier to separate large sequences of modules into distinct pathways.

The number of modules or structures per layer is 1.3 which shows networks are relatively thin. The number of modules is 20.6. The number of modules is much larger than both the MNIST and CIFAR-10 averages. We hypothesis that the GA learns to create separate pathways and so needs to essentially create two networks which doubles its size. However, the networks have only 1.6 unique modules per *main structure* which shows that this population of networks also favors reusing modules across the whole network.

The average kernel size, stride and channels is 5.8, 1.2 and 25 respectively. The number of channels is much lower than the other two image recognition problems. This problem is much less complex and so the number of feature detectors is correspondingly lower. The stride and kernel size are similar to the other two problems.

Concatenation was used 91% of the time, pooling was used 12% of the time. Weights were only used 12% of the time and biases were used only 41%. There were no FC modules in the *main structures*. These values are similar to the values of CIFAR-10 networks.

We observed that the best network evolved through several stages of development.

At the first stage, the network evolved into a one layer network with two structures as Figure 4.13 shows. Both structures contained 10 modules in sequential order. Each structure projected to a separate label – evolving separate independent structures. This demonstrates parallelization of the task by splitting the processing into two distinct streams. Upon further training, the network subdivided into two layers of two structures each. The structures were themselves five layers long of one module each. The earlier structures shared output to the next two structures and the final two structures projected to separate labels. Figure 4.14 shows the network at its final stage at 34 generations.

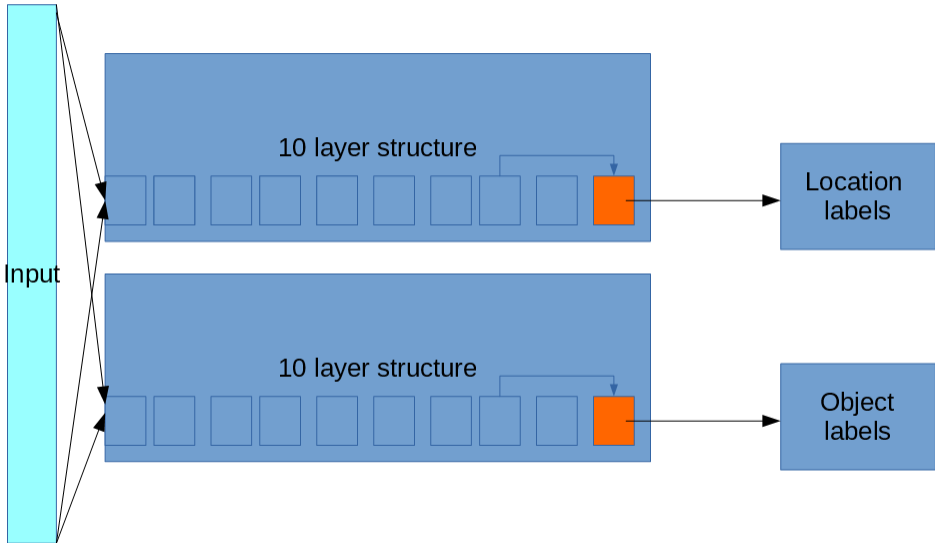

Figure 4.13: Early on in training, the network splits into two separate structure groups. One for each of the labels: location and object.

The final network demonstrates how low level features are shared between both labels then on the second layer the features "specialize" to handle different labels. This is similar to how an efficient lower-level feature representation is shared in the brain and then processed in different sections later on [8, 11]

An interesting observation is that the network evolved a hierarchical structure utilizing repeated structures with their own internal 5 layer networks. Figure 4.15 shows the networks' loss versus their depth after 34 generations. We see that the lowest loss is reached by networks with a depth of one. This demonstrates that certain problems may be more amenable to hierarchical processing and others may not be — such as MNIST and CIFAR-10.

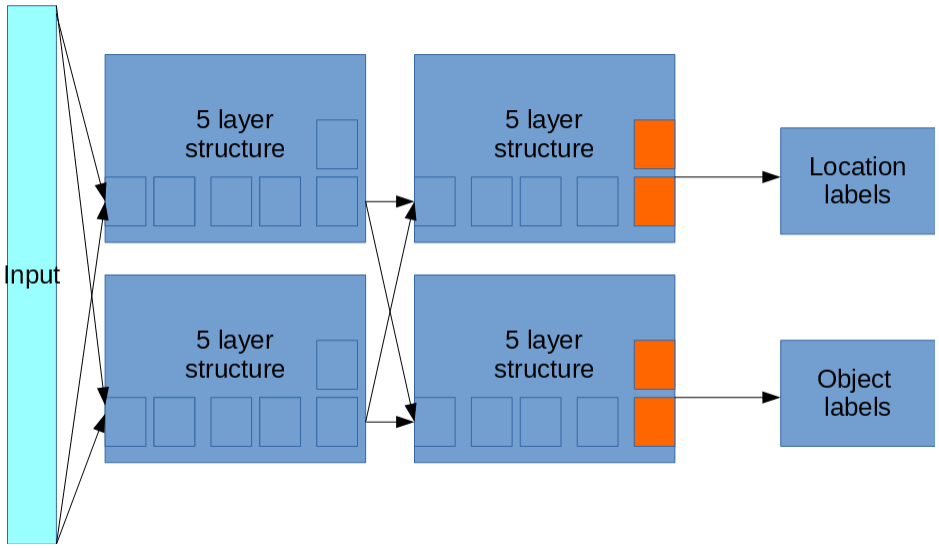

Figure 4.14: Best evolved network on the two class problem. It is a hierarchical network with each structure containing 5 layers. The orange modules denote output.

The results of the segmented task were able to achieve 100% accuracy after only 5 generations. The problem is simple and therefore to be expected. On more difficult problems, more generations would likely be required to find an optimal structure. However, as the generations increased, the loss continued to decrease which led to the creation of the final structure.

4.1 Discussion

We find that the use of jagged arrays overall works well. The approach is able to learn effective representations of networks that perform well across three different datasets and was able to quickly learn hierarchical representations for the 2-class problem.

Although, we did not do a direct comparison to graph-based architectures, we can compare some similarities and differences. Graph structures are comparable with jagged array structures in that they both can recreate any type of hierarchical feed-forward network. We would argue that mutations are more difficult to devise and employ in graph-based architectures than jagged arrays. Mutating a whole array is simple and fast, whereas a graph must be traversed and replaced. However, with graph-based structures, they are easily able to create multiple independent paths to a node, whereas with a jagged array architecture, it requires a hierarchy of structures

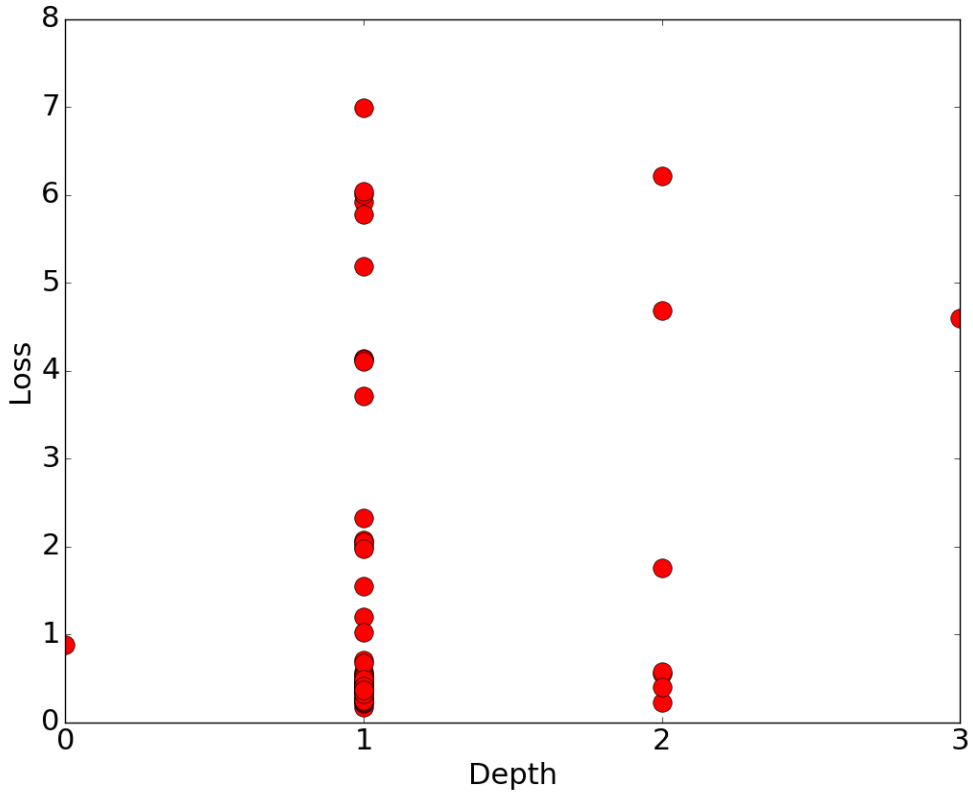

Figure 4.15: This figure plots the loss of each network after 34 generations on the two-class problem by their depth. The loss is lowest at depth one and two demonstrating that depth was necessary for lower loss.

to accomplish the same thing. We would like to explore future work on comparing the speed of mutation for graphs versus jagged arrays.

From the accuracies in the related work, we find that graph-based architectures exceeded the accuracy of the jagged array architecture. However, we can't say definitively that it is due to the representation scheme. After all, other comparable works used many more GPUs, sometimes 500+, to reach their results [29, 47]. If we had access to larger GPU clusters, we could test whether it also performed as well. Certainly, the final networks resembled other evolved networks for CIFAR-10, in that they shared common kernel sizes, channel outputs and number of layers and reuse. We would also like to try training networks using different GA parameters such as using tournament selection, using validation accuracy instead of loss as the fitness and experimenting without using crossover. As it stands, we can't recommend this

approach over existing approaches, however with the aforementioned changes above, we believe this approach deserves more consideration. Also, many of the proposed approaches share common characteristics, and indeed, our approach employs similar techniques to others such as evolving modules and structures separately and having the ability to substitute these modules into the *main structure* [31] and building structures into hierarchies [29]. So it remains to be seen what will emerge as the best combination of techniques for architecture search.

Finally, there is intrinsic value to the creation of alternative algorithms (in this case the use of jagged arrays) to accomplish a task even if they do not yet perform better than the state of the art. There have been many examples in the Deep learning field of researchers taking old algorithms, applying them on challenging problems with modern hardware and achieving extraordinary results. For example, Long Short Term Memory [18] was an older algorithm that was not used much until the advancements in GPU hardware acceleration and a few other advancements led to it being used in many applications today. We cannot say that the same situation will occur for our approach. However, this remains a first step in creating an algorithm for future exploration of its capabilities.

We were surprised that hierarchies did not develop in CIFAR-10. We thought that structures would evolve naturally, similar to how other popular networks used repeated convolution patterns at each layer [17]. We hypothesis that one reason is that the GA simply did not have enough generations to fully develop interesting sub structures and only had enough time to construct repeating modules across the layers. However, we did find sub-structures develop quickly on the two-class problem — this again could be due to the simplified problem space versus the much more complex CIFAR-10 problem. If not for complexity reasons, one hypothesis is that the 2 class problem explicitly required two streams of processing as the "where" and the "what" problems were very different. On the other hand, CIFAR-10's problem requires all parts of the network to contribute to the problem so sharing that information is critical for success.

The top hand-designed networks [43, 17] have all exhibited characteristics that the GA evolved. Our results confirm that heuristics developed over the past several years such as using shallow, long and thin networks, large output channels, kernel

sizes between 3 and 6 and reusing modules are beneficial. These characteristics all developed independently in the GA. In some ways, the best performing traditional networks had been crowd-sourced as researchers shared what they found worked well and then others made small changes and shared their results. The sharing of networks and tweaking their parameters can be thought of as a search in itself.

We find that the networks of different sizes (# of parameters, # of layers) performed almost as well as larger networks. This demonstrates one of the strengths of CNNs which acts as a universal approximator and without any fine-tuning can perform quite well on difficult image recognition tasks. However, the networks that achieved the very best performance, were those adapted specifically to the problem and the three top networks for each problem were very different from each other. The difference in the top networks shows that although the same network is particular good at adapting to many different datasets, there is a definite benefit to certain architectures over others.

We ran our GA over only a limited number of generations. We can't say what would develop over more generations as the computational resources are not at a sufficient capacity to allow us to run hundreds of generations yet. For future work, we will be able to test whether our findings remain consistent or if hierarchical structures do develop when the GA is trained for orders of magnitude longer. There are much larger networks that have been handcrafted with over 100 layers [17] but our GA was not given sufficient time to develop. In the future, we may be able to run the GA for longer and see if it develops deeper networks.

Another interesting finding is that the networks did not evolve a fully connected layer at the end of the network. In most published networks, the networks end with a fully connected layer. For all three problems, our best performing networks did not. This fact could be due to the evolutionary algorithm not being able to reproduce it or that the final layer was not as beneficial as previously thought. When comparing with the related work, we did not find that other evolved networks produced fully connected layers at the end of their networks either. This is most likely due to the previous convolutional layers developing a sufficient non-linear representation of the data which is linearly separable for a linear softmax layer.

We experimented briefly without using regularization on the size of networks.

Without size constraints the networks quickly grew to tens of millions of parameters due to specific mutations allowing networks to double in size. The large networks caused out of memory exceptions. So we first implemented a maximum limitation, but we found that the networks tended to grow to the maximum size very quickly and this would slow the discovery of efficient, small networks as the GA was required to test many large networks when a smaller network could perform just as well. At the same time, we did not want to restrict the size permanently so we determined the best approach was to gradually allow the network's to increase in size. We found that the gradual increase in size forced the GA to first consider networks of efficient structures before a larger size and sped up training time. The regularization effectively constrains the search space which leads to faster convergence to a solution.

Normally, a convolutional layer requires the preceding layer's output channels to be a uniform size. However, our approach allows for a set of modules on the preceding layer to produce different sized output "images". Other networks, like the Inception network [43] or ResNet [17], employ a "concatenation" or "pooling" layer to standardize the modules on the same layer. In our approach, we make no such requirement. The different sized outputs can either be automatically concatenated such as in those past networks or they can be preserved for successive layers up to the final layer. We were interested in determining whether networks would make use of this capability or use the technique of standardizing the output images. From our results, we found that both were used. In the MNIST problem, the two separate modules at layer 2 were not concatenated meaning they were convolved over separately by the two modules at layer 3. The number of parameters at layer 3 doubled as a result. The resulting output images were the same size, so they could have been concatenated, but we saw several networks that resulted that had several different sized outputs to exist through multiple layers. In the CIFAR-10 problem, the modules of the best performing network were all concatenated. For future work, we would like to investigate the underpinnings behind the benefit of carrying different sized outputs through the network and if there is a benefit to maintaining different sized outputs. Certainly, networks like DenseNet [19] have shown that projecting context from earlier layers to later layers is useful, but they employed a method of pooling at specific points to maintain uniformity of image size across the layer.

Table 4.1: Most occurring mutations in the *main structure* of the genome population.

Mutation Name	**Amount the mutation that occurred**
matrix creation copy row place below	151
matrix overwrite all with item in matrix	132
matrix creation copy double size vertically	131
matrix random row set standardized true	76
matrix creation layer append last structure	63
matrix creation append module	63
matrix creation copy row place above	63
matrix overwrite col with col item	63
structure add random identity skip connection	59
matrix creation copy double size horizontally	48
matrix overwrite weights non recursive	19
matrix overwrite col with random module	18
matrix creation copy module or structure on row	18
structure add random convolution skip connection	16
matrix overwrite row with row item	13
matrix overwrite row with single random module	13
matrix overwrite row with bias	12
matrix overwrite randomly with module	12
matrix remove row all but one module or structure	11
matrix creation copy row place horizontally	10
matrix random row set pooling true	10
matrix remove module or structure	9
matrix creation layer append first module	9
matrix overwrite randomly with structure	8
matrix overwrite row with weight	6
matrix random row set pooling false	6
matrix creation layer insert structure	4
matrix creation append structure	3
matrix overwrite row with single random structure	3
matrix remove row	3
matrix overwrite all with structure	2
matrix overwrite biases non recursive	2
matrix remove all	2
matrix creation layer insert module	2
matrix overwrite row with random module	2
mutate expression	1
matrix random row set standardized false	1
matrix overwrite col with random structure	1

Chapter 5

Conclusion

We conclude the thesis with a summary of the results, a discussion of the limitations and provide a glimpse at potential future work.

5.1 Summary

Traditionally, heuristics were used to create neural network architectures and they have proven successful in the past [24]. Architecture search algorithms did not become popular until hardware advancements made them computationally feasible. Now with the advent of architecture search algorithms, they have led to new advances in performance on benchmark datasets like CIFAR-10 [2, 47, 36].

We extended the research in this field by developing a novel structural scheme and used a genetic algorithm to evolve the structure of a convolutional neural network. In this way, we performed architecture search to find the best architecture suited to the individual problem.

In this thesis, we presented our genetic algorithm approach, we demonstrated how there are two phases of learning, the GA evolving the structure and then the training of the network on the dataset. We showed how the structures are organized and how they are transformed into TensorFlow graphs. Then we showed that our approach performs well on MNIST, CIFAR-10 and the 2-class problem. It is able to achieve respectable results on each benchmark. We observed the best performing networks and found that they exhibited unique traits among all three showing that the network evolves structures for each problem. We observed that the GA evolved the heuristics that are used currently on hand-crafted models such as the re-use of modules and multiple modules per layer.

5.2 Threats to Validity

In this section, we describe possible threats of validity to our study.

We could not perform an extensive statistical evaluation of the genetic algorithm's performance. A 10 fold cross validation was not possible with our current hardware. However, heuristically, we found that our approach was capable of learning across at least three different problems demonstrating its ability to handle different datasets. We also found after running several short runs of a dozen generations that the average loss of the genetic algorithm was reduced. Additionally, in the literature, some of the seminal papers in Deep Learning never report exhaustive analyses. The lack of analyses is due in part to the length of time Deep Learning algorithms take to train (ours took over 3 weeks). This means that traditional statistical analysis is not feasible. Researchers let the average accuracy on the test data serve as the quantitative benchmark of the algorithm.

Genetic algorithms are not guaranteed to converge to the solution and one of its problems is that they get stuck in local minimum. Although this is true, we employ elitism to at the very least, not create worse solutions. Secondly, we found that the networks that were evolved consistently matched the benchmarks of other solutions, highlighting its ability to at the very least reach the level of past network architectures.

Although we tested three datasets (MNIST, CIFAR-10 and 2-class problem), we cannot say that the approach generalizes to all datasets. Future work is required to test different datasets to see what architectures result and if the GA can successfully evolve a CNN capable of performing well on other problems.

Our approach relies on selecting parameters for our network such as the learning rate. We used our own heuristics to determine the best hyper-parameters for the problem. Where possible we used knowledge from the literature to determine the parameter values. At the same time, even if the hyper-parameters are sub-optimally chosen, the GA is able to work within the constraints of the hyper-parameter space to evolve networks that perform well with those given hyper-parameters.

Our approach requires creating different types of mutations. We do not know if there are favorable mutations that have been left out or if there are mutations that

hinder the progress of the GA. However, the nature of the GA is that each mutation has only a small chance of occurring at any given generation and network. There is a strong chance that network individuals will carry over between generations without a specific mutation occurring. Therefore, even if we included a bad mutation operator, the GA could effectively ignore it by selecting the strong performing networks without that mutation. Future work is required to perform an extensive analysis on which mutations are most beneficial and which can be excluded. We felt that adding as many as possible allowed the GA the most freedom.

5.3 Future Work

For future work, we would like to run our approach on more powerful hardware, that means more GPUS. Genetic algorithms are inherently parallelizable and so the task of adding more GPUs to process each network should not be too difficult. Given more processing power, more generations of the GPU can be processed which means a chance for more complex structures to evolve and better structures for the problem at hand. Some of the other techniques used up to 200+ GPUs compared to our one GPU [45].

We would also like to experiment with different genetic algorithm techniques. There are many different types of crossover and selection techniques like tournament selection. It remains to be seen if the other techniques would have faster learning rates or higher accuracies. We would also like to explore different datasets to see the different structures that result.

We would like to experiment with using fitness sharing which helps preserve diversity in the population [30]. Additionally, we would consider using evolutionary multi-objective optimization through Pareto dominance to automatically rank candidates by their complexity and accuracy rather than resort to a manual weighting between accuracy and complexity [3].

Appendices

Appendix A

Pseudo Code

Pseudo code for some of the algorithms developed for this problem. Certain parts have been simplified in order to convey the basic theory without bogging down in the complexities of the implementation itself.

Algorithm 1: fc_input — Pseudo-code for recursively connecting the graph to the input.

```
Data: x, element, data
if element is a Module then
    W ← weight_variable([kernel.x = element.kernel, kernel.y =
      element.kernel, in = data.num_channels, out = element.channel_output];
    b ← bias_variable(element.channel_output);
    h ← relu(matmul(x, W) + b);
    Add batch normalization and dropout;
    if element.pooling then
        out_height ← int(math.ceil(out_height/2.0));
        out_width ← int(math.ceil(out_width/2.0));
    end
    graph_pointer ← Pointer(h, out_height, out_width);
    element.graph_pointers.append(graph_pointer);
else if element is a Structure then
    if len(element.matrix) > 0 then
        for element in element.matrix[0] do
            fc_input(x, element, data);
        end
    end
end
```

Algorithm 2: fc_structure — Pseudo-code for recursively creating a connected Tensorflow graph of the *main structure*. Starting at the second layer, iterate over each module and connect with the previous layer's modules. Concatenate modules if module calls for it.

```
Data: structure
for for i, layer in enumerate(structure layers) do
    if i > 0 then
        modules ← find_connected_previous_modules(structure[i − 1]);
        if i >= 2 then
            modules ← add_any_skip_connections();
        end
    end
    for j, element in enumerate(layer) do
        if element.concatenate is True then
            concatenated_modules ← concatenate(modules);
            connect_modules(element, concatenated_modules);
        else
            connect_modules(element, modules);
        end
        if element is a Structure then
            fc_structure(element);
        end
    end
end
```

Algorithm 3: connect_modules — Pseudo-code for recursively connecting a module to its connected modules from previous layers.

```
Data: next_element, previous_modules
if next_element a Module then
    next_module ← next_element;
    for previous_module in previous_modules do
        for previous_pointer in previous_module.graph_pointers do
            out_height ← ceil(previous_pointer.img_height /
             next_module.stride);
            out_width ← ceil(previous_pointer.img_width / next_module.stride);
            W ← weight_variable([kernel.x = element.kernel, kernel.y =
             element.kernel, in = previous_module.channel_output, out =
             next_module.channel_output];
            b ← bias_variable(next_module.channel_output);
            h ← relu(matmul(previous_pointer.graph_node, W) + b);
            Add batch normalization and dropout;
            if element.pooling then
                out_height ← int(ceil(out_height/2.0));
                out_width ← int(math.ceil(out_width/2.0));
            end
            graph_pointer ← Pointer(h, out_height, out_width);
            next_module.graph_pointers.append(graph_pointer);
        end
    end
else if next_element is a Structure then
    if len(next_element.layers) > 0 then
        for module in next_element.first_layer do
            connect_modules(module, previous_modules);
        end
    end
end
```

Algorithm 4: concatenation — Pseudo-code for concatenating a set of modules into a single module

```
Data: modules
Result: new_cnn_module
instantiate new_cnn_module;
for module in modules do
    if module is Module then
        smallest_height, smallest_weight ← get_min(module.graph_pointers);
    end
end
for module in modules do
    for previous_pointer in module.graph_pointers do
        new_cnn_module.channel_output += module.channel_output;
        if previous_pointer.img_height > smallest_height then
            new_graph_node ← downsample(previous_pointer, smallest_height,
             smallest_width);
            same_size_graph_nodes_cnn.append(new_graph_node);
        else
            same_size_graph_nodes_cnn.append(previous_pointer.graph_node);
        end
    end
end
concatenated_cnn ← concatenate(same_size_graph_nodes_cnn, 3);
pointer ← Pointer(concatenated_cnn, smallest_height, smallest_width);
new_cnn_module.graph_pointers.append(pointer);
return new_cnn_module;
```

Algorithm 5: connect_output — Pseudo-code for connecting the output with the *main structure's* output in TensorFlow.

```
Data: main structure, output_size, output_size
Result: y
output_pointers, num_nodes_array ← get_output_modules(structure);
num_nodes ← sum(num_nodes_array);
if num_nodes > 0 then
    output concatenate(output_pointers, 1);
    W ← weight_variable(num_nodes, output_size);
    b ← bias_variable(output_size);
    Add batch normalization and dropout;
    y ← matmul(output, W) + b;
    return y;
else
    return None;
end
```

Algorithm 6: crossover_structure — Pseudo-code for crossover of two structure arrays

```
   Data: gene_a, gene_b
 1 structure_a, structure_b = gene_a.structures, gene_b.structures if
    len(structure_a) <= len(structure_b) then
 2 |   min_len ← len(structures_b);
 3 else
 4 |   min_len ← len(structures_;
 5 end
 6 for i in range(min_len) do
 7 |   if random_number < mod_constants.crossover_rate then
 8 |   |   temp ← structures_a[i];
 9 |   |   structures_a[i] ← structures_b[i];
10 |   |   structures_b[i] ← temp;
11 |   |   switch elements of structures_a[i]'s jagged_array with corresponding
   |   |    elements from gene_b;
12 |   |   switch elements of structures_b[i]'s jagged_array with corresponding
   |   |    elements from gene_a;
13 |   |   replace references to structures_b[i] with references to structures_a[i];
14 |   |   replace references to structures_a[i] with references to structures_b[i];
15 |   end
16 end
```

Algorithm 7: crossover_modules — Pseudo-code for crossover of two module arrays

```
Data: gene_a, gene_b
modules_a, modules_b = gene_a.modules, gene_b.modules;
if len(modules_a) <= len(modules_b) then
    min_len ← len(modules_a);
else
    min_len ← len(modules_b);
end
for i in range(min_len) do
    if random_number < mod_constants.crossover_rate then
        temp ← modules_a[i];
        modules_a[i] ← modules_b[i];
        modules_b[i] ← temp;
        replace gene_b weights of modules_a[i] with corresponding weights from
         gene_a;
        replace gene_a weights of modules_b[i] with corresponding weights from
         gene_b;
        replace references to modules_a[i] in gene_b with modules_b[i];
        replace references to modules_b[i] in gene_a with modules_a[i];
    end
end
```

Algorithm 8: Dual output Stream — Pseudo-code for splitting the structure's output into the 2 two-class softmax outputs.

```
Data: a_structure
Result: y_pos, y_what
output_pointers, num_nodes_array = get_output_modules(a_structure);
if output_pointers > 1 then
    half_way ← len(output_pointers)/2;
    pos_outputs ← output_pointers[0:half_way];
    pos_num_nodes ← sum(num_nodes_array[0:half_way]);
    what_outputs ← output_pointers[half_way:];
    what_num_nodes ← sum(num_nodes_array[half_way:]);
    y_pos ← do_output_calc(pos_outputs, pos_num_nodes, len(y_pos));
    y_what ← do_output_calc(what_outputs, what_num_nodes, len(y_what));
else
    output_pointers, num_nodes_array ← get_output_modules(structure);
    num_nodes ← sum(num_nodes_array);
    y_pos ← do_output_calc(output_pointers, num_nodes, len(y_pos));
    y_what ← do_output_calc(output_pointers, num_nodes, len(y_what));
end
return y_pos, y_what;
```

Appendix B

Mutation Descriptions

Structure Array:

Create new structure: Creates a new empty structure and adds it to the structure array.

Create a new copy structure: Creates a new copy of an existing structure and adds it to the structure array.

Merge two structures side by side: Create a new structure and place two other random structures inside the new structure's jagged array side by side.

Merge two structures on top: Create a new structure and place two other random structures inside the new structure's jagged array on two different layers.

Merge two modules side by side: Create a new structure and place two random modules inside the new structure's jagged array side by side.

Merge two modules on top: Create a new structure and place two other random modules inside the new structure's jagged array on two different layers.

Swap positions of structure: Swap positions of two random structures in the structures array.

Swap starting position: Swap the structure at the starting position in the structures array with another random structure. Essentially replacing the *main structure* with another structure in the structures array.

Shift structure left: Swap positions with a structure to the left of a random structure.

Shift structure right: Swap position with a structure to the right of a random structure.

Cut and rearrange: Split the structures array in two and then place the second half before the first.

Remove structure: Remove random structure from structures array

Transfer structure (keep old): All references to a random structure are transferred

to another random structure and the references are converted to the new structure. The original structure is kept in the array.

Transfer structure (remove old): All references to a random structure are transferred to another random structure and the references are converted to the new structure. The original structure is removed from the array.

Weights:

Adjust value: Add normal random variable value to the weights array.

Double value: Double the current value of a weight element.

Divide value: Divide current value of a weight element by two.

Create new weight: Create a new weight and add it to the weights array.

Create a new copy of weight: Create a new weight that is a copy of an existing weight initialization and add to the weights array.

Transfer weights (keep old): All references to a random weight are transferred to another random weight — converting them to the new weight. The original weight is kept in the weight array.

Transfer weights (remove old): All references to a random weight are transferred to another random weight — converting them to the new weight. The original weight is removed from the weight array.

Remove weight: Remove the weight initialization from the weight array.

Biases:

Adjust value: Add normal random variable value to the biases array.

Double value: Double the current value of a bias element.

Divide value: Divide current value of a bias element by two.

Create new bias: Create a new bias initialization and add it to the biases array.

Create a new copy of a bias: Create a new bias that is a copy of an existing bias initialization and add to the biases array.

Transfer biases (keep old): All references to a random bias are transferred to another random bias — converting them to the new bias. The original bias is kept in the bias array.

Transfer biases (remove old): All references to a random bias are transferred to another random bias — converting them to the new bias. The original bias is removed from the bias array.

Remove bias: Remove the bias initialization from the bias array.

Structure's Jagged Array Mutations):

Insert module as a new layer (first): Insert a random module as a new layer at the beginning of the jagged array.

Insert module as a new layer (last): Insert a random module as a new layer at the end of the jagged array.

Insert a module as a new layer (anywhere): Insert a random module as a new layer between random existing layers.

Insert module on exiting layer: Insert a random module on a random layer.

Insert structure as a new layer (first): Insert a random structure as a new layer at the beginning of the jagged array.

Insert structure as a new layer (last): Insert a random structure as a new layer at the end of the jagged array.

Insert a structure as a new layer (anywhere): Insert a random structure as a new layer between random existing layers.

Insert structure on exiting layer: Insert a random structure on an existing random layer.

Create copy of an element in a layer and insert: Create a copy of an element in a random layer (could be a module or structure) and add it to the same layer.

Double size horizontally: Copy all the elements in the jagged array and add it horizontally to the jagged array — effectively doubling the size.

Double size vertically: Copy all the elements in the jagged array and add it vertically to the jagged array — effectively doubling the size.

Copy layer elements, append to the same layer: Copy all of a layer's elements and append them to the same layer.

Copy layer elements add as a new layer (below): Copy a layer then insert the new copy to the jagged array below the original layer.

Copy layer elements, add as a new layer (above): Copy a layer then insert the new copy to the jagged array above the original layer.

Overwrite a layer with a random module: Overwrite a layer with a random module. Older elements are replaced by the random module.

Overwrite a layer with a layer module or structure: Overwrite a layer with either a layer or structure randomly chosen from the jagged array.

Overwrite a column with a random module: Select a column index in the jagged array and replace each layer at that column index with a random module.

Overwrite a column with a random structure or module: Select a column index in the jagged array and replace each layer at that column index with a random module or structure in the jagged array.

Overwrite all elements with a module: Randomly choose a module from the modules array and overwrite all elements in the jagged array with the chosen module.

Overwrite all elements with an structure or module: Randomly select a module or structure from the jagged array and overwrite all the elements with it.

Overwrite a layer with a weight: Choose a random weight to overwrite a layer's weights.

Overwrite a layer with a bias: Choose a random bias and overwrite a random layer's biases.

Overwrite a random element with a module: Overwrite a random element with a random module from the modules array.

Overwrite a random element with a structure: Overwrite a random element with a random structure from the structures array.

Overwrite biases of elements (non-recursive): Overwrite the biases of elements in the jagged array non-recursively.

Overwrite weights of elements (non-recursive): Overwrite the weights of elements in the jagged array non-recursively.

Overwrite a layer with a random structure: Overwrite a random layer with a random structure.

Overwrite a column with a random structure: Overwrite a random column in the jagged array with a random structure.

Overwrite all with structure: Select a random structure from the structures array

and overwrite all elements of the jagged array with it.

Remove jagged array layer: Delete a random layer.

Remove jagged array module or structure: Randomly select an element from the jagged array and remove it.

Remove all: Empty the jagged array of all elements.

Overwrite a layer with a single random module: Random layer is reduced to a single random module.

Set a module's pooling to true: Randomly select a module in the jagged array and set its pooling to true.

Set a module's pooling to false: Randomly select a module in the jagged array and set its pooling to false.

Set a module's concatenation parameter to true: Set a module's concatenation parameter to true

Set a module's concatenation parameter to false: Set a module's concatenation parameter to false

Overwrite a layer with a single random structure: Replace a random layer with only a single random structure.

Remove all but one element from layer: Remove every element from a random layer except one (which is selected randomly).

Add random skip connections: Add a skip connection to the structure at random point. There must be a layer in between the skip connection layers so the number of layers must be 3 or greater.

Individual Modules:

Change a module's weight initialization: Set module's weight initialization to a random weight initialization from the weights array.

Change a module's bias initialization: Set module's bias initialization to a random bias initialization from the biases array.

Add 1 node to fully connected module: Add 1 node to the fully connected module.

Subtract 1 node of a fully connected module: Subtract 1 node from the fully connected module.

Double number of nodes: Double the number of nodes from the fully connected module.

Halve number of nodes: Halve the number of nodes from the fully connected module.

Add 1 output channel to CNN: Add 1 output channel to the CNN module.

Subtract 1 output channel to CNN: Subtract 1 output channel to the CNN module.

Double output channels: Double the number of output channels on the CNN module.

Halve output channels: Halve the number of output channels on the CNN module.

Add 1 to kernel: Add 1 to kernel size of the CNN module.

Subtract 1 to kernel: Subtract 1 to kernel size of the CNN module.

Add 1 stride in CNN module: Add 1 to stride in the CNN module.

Subtract 1 stride in CNN module: Subtract 1 from stride in the CNN module.

Flip pooling: Flip the true false flag of the CNN module.

Flip concatenation: Flip the true/false flag of the CNN module.

Modules Array:

Create new CNN module: Create a new convolutional module and add to modules array.

Create new FC module: Create a new fully-connected module and add to modules array.

Create module copy: Add a copy of an existing module.

Transfer modules (keep old): Transfer references of one module to another module and convert the references to the new module. Keep old un-referenced module in array.

Transfer modules (remove old): Transfer references of one module to another module and convert the references to the new module. Remove old un-referenced module in array.

Swap modules: Swap two random modules in the modules array

Shift modules left: Swap a random module to its left neighbour's position.

Shift modules right: Swap a random module to its right neighbour's position.

Remove module: Remove random module in modules array.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Filipe Assunçao, Nuno Lourenço, Penousal Machado, and Bernardete Ribeiro. Denser: Deep evolutionary network structured representation. *arXiv preprint arXiv:1801.01563*, 2018.

[3] Khaled Badran and Peter I Rockett. The influence of mutation on population dynamics in multiobjective genetic programming. *Genetic Programming and Evolvable Machines*, 11(1):5–33, 2010.

[4] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[5] Rodrigo Benenson. Classification datasets results, 2016. [Online; accessed 9-March-2018].

[6] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[7] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[8] Edward M Callaway. Structure and function of parallel pathways in the primate early visual system. *The Journal of physiology*, 566(1):13–19, 2005.

[9] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[10] Omid E David and Iddo Greental. Genetic algorithms for evolving deep neural networks. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 1451–1452. ACM, 2014.

[11] James J DiCarlo, Davide Zoccolan, and Nicole C Rust. How does the brain solve visual object recognition? *Neuron*, 73(3):415–434, 2012.

[12] Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A. Rusu, Alexander Pritzel, and Daan Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *CoRR*, abs/1701.08734, 2017.

[13] Kunihiko Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.

[14] Melvyn A Goodale and A David Milner. Separate visual pathways for perception and action. *Trends in neurosciences*, 15(1):20–25, 1992.

[15] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

[16] David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[18] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[19] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[20] Robert A Jacobs, Michael I Jordan, and Andrew G Barto. Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks. *Cognitive science*, 15(2):219–250, 1991.

[21] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12. ACM, 2017.

[22] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[23] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[24] 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.

[25] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

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

[27] Frank Hung-Fat Leung, Hak-Keung Lam, Sai-Ho Ling, and Peter Kwong-Shun Tam. Tuning of the structure and parameters of a neural network using an improved genetic algorithm. *IEEE Transactions on Neural networks*, 14(1):79–88, 2003.

[28] Chenxi Liu, Barret Zoph, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. *arXiv preprint arXiv:1712.00559*, 2017.

[29] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017.

[30] Robert I McKay. Fitness sharing in genetic programming. In *Proceedings of the 2Nd Annual Conference on Genetic and Evolutionary Computation*, pages 435–442. Morgan Kaufmann Publishers Inc., 2000.

[31] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving deep neural networks. *arXiv preprint arXiv:1703.00548*, 2017.

[32] Marvin Minsky, Seymour A Papert, and Léon Bottou. *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

[33] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. Evolutionary training of sparse artificial neural networks: a network science perspective. *arXiv preprint arXiv:1707.04780*, 2017.

[34] David J Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. In *IJCAI*, volume 89, pages 762–767, 1989.

[35] Nicolas Pinto, David Doukhan, James J DiCarlo, and David D Cox. A high-throughput screening approach to discovering good forms of biologically inspired visual representation. *PLoS computational biology*, 5(11):e1000579, 2009.

[36] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

[37] Beyond Regression. *New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Ph. D Dissertation, Harvard University, Department of Applied Mathematics, 1974.

[38] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[39] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[40] Chris Schumacher, Michael D Vose, and L Darrell Whitley. The no free lunch and problem description length. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, pages 565–570. Morgan Kaufmann Publishers Inc., 2001.

[41] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[42] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012.

[43] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, page 12, 2017.

[44] Lingxi Xie and Alan Yuille. Genetic cnn. *arXiv preprint arXiv:1703.01513*, 2017.

[45] Zhao Zhong, Junjie Yan, and Cheng-Lin Liu. Practical network blocks design with q-learning. *arXiv preprint arXiv:1708.05552*, 2017.

[46] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[47] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2017.