UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

# EFFICIENT NEURAL ARCHITECTURE SEARCH USING A GENETIC ALGORITHM

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

BY

BRANDON S. MORGAN
Norman, Oklahoma
2022

EFFICIENT NEURAL ARCHITECTURE SEARCH USING A GENETIC ALGORITHM

A MASTER'S THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Dean Hougen

Dr. Dimitrios Diochnos

Dr. Chongle Pan

# Abstract

NASNet and AmoebaNet are state-of-the-art neural architecture search systems that were able to achieve better accuracy than state-of-the-art human-made convolutional neural networks. Despite the innovation of the NASNet search space, it lacks the ability to express flexibility in terms of optimizing non-convolutional operation layers, such as batch normalization, activation, and dropout. These layers are hand designed by the architect prior to optimization, limiting the exploration possible for model architectures by narrowing down the search space. In addition, the NASNet search space can not allow for many non-classical optimization techniques to be applied as it lacks the ability to be expressed in a fixed-length, floating-point, multidimensional array. Lastly, both NASNet and AmoebaNet use an extensive amount of computation, both evaluating 20,000 models during optimization, consuming 2,000 GPU hours worth of computation. This work addresses these limitations by, first, changing the NASNet search space to include optimization of non-convolutional operation layers through the addition of a building block that allows for the optimization for the order and inclusion of these layers; second, proposing a fixed-length, floating-point, multidimensional array representation to allow other non-classical optimization techniques, such as particle swarm optimization, to be applied; and third, proposing an efficient genetic algorithm, while using state of-the-art techniques to reduce training com-

plexity. After only 1,300 models evaluated, consuming 190 GPU hours, evolving on the CIFAR-10 benchmark dataset, the best model configuration yielded a test accuracy of 94.6% with only 1.3 million parameters, and a test accuracy of 95.09% with only 5.17 million parameters, outperforming both ResNet110 and WideResNet. When transferring to the CIFAR-100 benchmark dataset, the best model configuration yielded a test accuracy of 71.1% with only 1.3 million parameters, and a test accuracy of 76.53% with only 5.17 million parameters.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Introduction

Convolutional neural networks are neural networks that are specifically designed for processing images. Since the emergence of AlexNet [17], many different architectures have been proposed by researchers in order to maximize performance. The most notable networks, utilized throughout this paper, are VGGNet [25], ResNet [12], Inception [30], Inception-ResNet, and Xception [29]. These architectures serve as the foundation from which most advanced convolutional neural networks stem. However, these human-made architectures are carefully designed based upon current research, and researchers rarely explore other variants or possibilities. As a result, manually exploring architectures can be an extremely time consuming process. With the goal of exploring other possible network architectures automatically, the domain known as neural architecture search (NAS) emerged. Arguably, the first recognized NAS system was [28], which utilized a genetic algorithm to evolve the architecture of a neural network for a reinforcement learning problem. Since NEAT, the most recognized NAS system applied to convolutional neural networks at large scale, arguably, is NASNet [38]. NASNet proposed a highly complex, yet powerful, search space from which scalable

networks could be created. NASNet utilized a recurrent neural network trained through reinforcement learning to explore the search space. Shortly after NASNet, AmoebaNet [22] emerged, which tackled the NASNet search space, except utilizing a mutation-only genetic algorithm. Both NAS systems were extremely successful, achieving better than state-of-the-art human-made models on two very prominent benchmark image dataset, CIFAR10 [16] and ImageNet [3]. Despite the success of the NASNet search space, it suffers from three primary issues. First, despite being a search space that explores possible architectures, it was static in terms of non-convolutional operational layers. In the NASNet search space, non-convolutional operation layers, such as batch normalization, activation, and dropout, were not allowed to be explored, but were preset by the user. As a result, this static formulation hindered the range from which possible model architectures could be created. Secondly, the NASNet search space lacked the ability to be expressed in terms of a fixed, continuous, floating-point multidimensional array, which most non-classical optimization techniques, such as particle swarm optimization, leap frog, hill climber, and low-level genetic algorithms, require to be applied. As a result, the representation of the NASNet search space prevented other non-classical optimization techniques from being applied and tested. Third, both NASNet and AmoebaNet used an extreme amount of computation, as both systems each consumed 2,000 GPU hours worth of computation while evaluating 20,000 models over the course of 7 days with 450 GPUs.

In this work, a number of proposals are described in order to lessen these three primary issues. First, in order to allow for the optimization of the non-convolutional operational layers, an extension to the NASNet search space was created for the ability to control the inclusion, or exclusion, and ordering between the non-convolutional operational layers. Second, in order to allow for other

non-classical optimization algorithms, such as particle swarm optimization to be applied, a new representation of the altered NASNet search space was proposed. Third, in order to reduce the computational requirements, the phenomenon of super convergence [26] was utilized in order to speed up the training of models at small projections. In addition, a few different variants of a genetic and particle swarm optimization were discussed and tested at small scale in order to examine the performance of the search algorithms when only 1,300 models were allowed to be evaluated.

The organization of this thesis adheres to the following layout. Chapter 2 introduces the necessary background information necessary to understand the foundation of this work in advanced convolutional networks and non-classical optimization techniques. Chapter 3 introduces the related work of AmoebaNet and NASNet in greater detail as a means to better understand the NASNet search space. Chapter 4 introduces the methodologies behind the creation of the After, the new representation, and the proposed genetic and particle swarm algorithms. Chapter 5 showcases the results from the experiments that were performed for comparing the chosen algorithms, as well as the full-scale optimization on the CIFAR10 dataset. Chapter 6 reviews over the results found in Chapter 5, expounding upon them in detail while also mentioning future work. Lastly, Chapter 7 summarizes the research, contributions of this work, and results.

# Chapter 2

# Background Knowledge

## 2.1 Introduction to Computational Intelligence

Computational intelligence (CI) is a sub-field of artificial intelligence (AI) devoted to modeling intelligence from biological inspirations. CI comprises of many different fields, namely artificial neural networks (ANN), evolutionary computation (EC), and swarm intelligence (SI). The inspiration behind ANNs is to model biological neural networks found in the brain for learning patterns and underlying functions. Both EC and SI seek to solve optimization functions; however, EC solves such problems while being inspired from biological evolution, while SI is inspired from swarm behavior. The process of applying evolution or swarm behavior for the optimization of ANNs in some capacity is referred to as neuroevolution (NE). NE is an exciting field and topic of AI as the idea of automatizing deep learning models has tremendous applications such as solving controller like environments [28].

In this chapter, the necessary background knowledge will be discussed in order to fully understand and grasp the research of this work. First, a brief

overview of machine learning and optimization theory will be discussed in order to understand the basics of supervised learning and optimization. Next, a deep dive into EC will be pursued in order to understand the evolutionary and swarm intelligent algorithms utilized throughout the work. On the other-end of the spectrum of CI, ANNs will be discussed along with their extension for image processing, convolutional neural networks (CNNs). Subsequently, the field of neural architecture search (NAS) will be discussed in order to understand the intersection between EC and CNNs.

## 2.2 Machine Learning

Machine learning is a branch of AI and computer science focused on learning representations from data. There are three main types of learning in machine learning: supervised, unsupervised, and reinforcement learning. In supervised-learning, the objective is to learn an underlying function to map input-output pairs. Supervised learning is mainly utilized for classification, when the output is discrete, or for function approximation, also known as regression, when the output is continuous. Unsupervised learning is learning when no defined output value is available; but instead, the objective is to learn some type of representation from the input. An example of an unsupervised learning task is clustering, where the objective is to group data points together based on some type of learned representation from the algorithm. Lastly, reinforcement-learning, like supervised learning, takes in a set of inputs; however, the outputs are no longer trying to be mapped, but instead, the outputs are now rewards where the goal is to maximize the said reward. In machine learning, a model is utilized to learn the representations and mapping in supervised, unsupervised, and reinforcement

learning. Because supervised learning is the only type of machine learning utilized in this work, it will be the only category discussed in further detail.

### 2.2.1 Supervised Learning

In supervised learning, machine learning models are trained upon a training data set, which consists of $n$ example input-output pairs, $(x_1, y_1), \ldots (x_n, y_n)$, where $x$ is the input and $y$ is the actual output, otherwise known as ground-truth. The objective is to find an underlying function, utilizing the machine learning model, to map the input to the ground truth in the training set.

For supervised learning algorithms, error, or loss, functions have been created in order to measure how good a model performs at a particular task. Machine learning models try to minimize the error of these loss functions in order to achieve maximum success at better representing the underlying function. Throughout the research of this work, primarily one very important loss function was utilized, cross entropy.

#### Cross Entropy

Cross-entropy is a type of loss metric utilized for classification type scenarios, when the output variable is discrete. Cross entropy, which originated from information theory, measures how well two probability distributions are similar concerning a random variable. For discrete distributions, cross entropy is described in Equation 2.1,

$$H(P, Q) = -\sum_{i=1}^{m} P_i \log_2(Q_i) \tag{2.1}$$

where $P$ represents the predicted distribution, $Q$ represents the actual distri-

bution, $m$ is the total number of classes, and $i$ is the $i^{th}$ partition representing the probabilities for the $i^{th}$ class from the distribution.

Most classification machine learning algorithms output a discrete probability distribution representing all the possible classes. Cross entropy works by summing up the probabilities of the actual class occurring for a given data instance, multiplied by the algorithms predicted probability. In this way, the objective for classification machine learning algorithms utilizing cross-entropy is to match the actual class distribution for all data instances. Even though the output from these algorithms is a probability distribution across all possible classes, the predicted class can easily be obtained by taking the largest probability amongst the predicted distribution to be the overall predicted class for a given data instance. However, cross entropy lacks the ability to extend applicable information in classification, as quantifying how well the predicted distribution matches the actual does not yield satisfactory application. To resolve this issue, simple accuracy can be utilized in order to measure how accurate the model is at predicting classes. In its simplest form, accuracy is defined to be the sum of the number of times the predicted class matches the output class over the total number of data instances utilized

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \tag{2.2}$$

### Overfitting and Underfitting

A common issue that arises with machine learning models is over-fitting, a phenomenon that occurs when the error on the training dataset is small but the error on another dataset, containing unseen data, is much larger. This secondary dataset is often called the test-set, which is used to test how well the machine

learning model is capable of generalizing to new unseen input data. For an example of overfitting, please refer to Figure 2.1. The blue line represents the actual trend, where both the blue and orange points are sampled from this trend with added noise; however, the blue points represent the training dataset while the orange points represent the testing dataset. An $8^{th}$ degree polynomial model as applied to the training dataset and its predicted regression curve was plotted in orange. One can see that the polynomial over-fitted the the training data and does not generalize well to the testing dataset as the mean squared error ($MSE$), a loss function for regression type problems, for the testing set is roughly 17.5 times larger than the training.

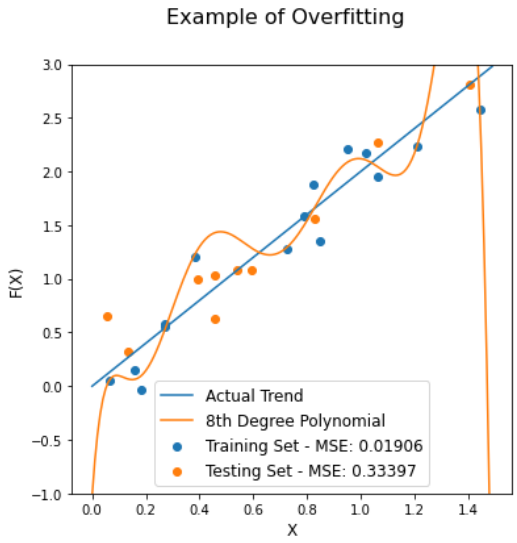



Figure 2.1: An example of overfitting.

Possible solutions to overfitting will be discussed later on in the section on artificial neural networks. In contrast to overfitting, underfitting occurs when the model is unable to learn an acceptable representation as the accuracy metric is extremely poor. Underfitting occurs from two particular scenarios, either the machine learning model is not complex enough to learn a representation, or when

the data is insufficient in terms of information across the variables. When the data is insufficient, little to nothing can be done on the side of the machine learning engineer; however, if the model cannot learn, then more complex models containing adjustable numbers of trainable parameters can be utilized in order to increase complexity.

### Cross-Validation

Machine learning algorithms can be extremely powerful at prediction; however, most algorithms have a set of hyper-parameters that need to be tuned in order to achieve acceptable results. Hyper-parameters are parameters that are not learned by the model, but instead are set by the user in order to affect the learning and algorithmic process. Because the success of these algorithms greatly depends on the initialization of these hyper-parameters, there have been established methodologies on how to choose and select such parameters. The simplest methodology is to try out different sets and combinations of hyper-parameters for the selected machine learning algorithm by training each unique combination on the training dataset, and then comparing their generalization on the testing set. Although this process seems fool-proof, by comparing the algorithms on the testing set, the machine learning engineer is inadvertently starting to bias towards the test dataset, which is supposed to be completely unseen data instances for evaluation. A simple way to bypass this problem is to create a secondary test set, known as the validation dataset. The choices of hyper-parameters are again trained on the training dataset, but then are compared for generalization on the validation dataset, and then the best combination from the validation set is selected to measure its success against the test dataset. The process of comparing different hyper-parameters or models is known as cross-validation.

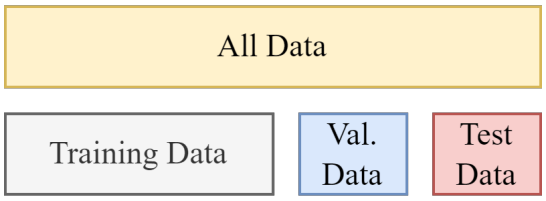



Figure 2.2: Hold-Out Method for Cross-Validation

This methodology of selecting and creating particular training, validation, and testing datasets is commonly referred to as the hold-out method, where a subset of the data is held out for each particular phase of training, comparison, and evaluation, as depicted in Figure 2.2. In practice, when the amount of data is minimal, utilizing a hold-out method is ineffective as the method only works when the sample size is large enough for each hold-out set to resemble the population. K-fold cross-validation is a sampling procedure that alleviates this problem by training the model a total of K times, each on a different fold of the dataset. The entire dataset is split into a training and testing set, and the training set is split into K folds, where each fold is sub split into a training and validation set. The average error metric across all validation sets from all K folds is then utilized to compare the hyper-parameters, while the best is selected for evaluation on the test set, as depicted in Figure 2.3.

Another common necessity in machine learning is comparing different machine learning models. The same process described for selecting hyper-parameters above can be utilized for comparing models: train each model on the training set, compare using the validation set, and then select the best from the validation set and measure its performance against the test set. The same process can also be utilized for K-fold cross validation when the amount of data is insufficient.

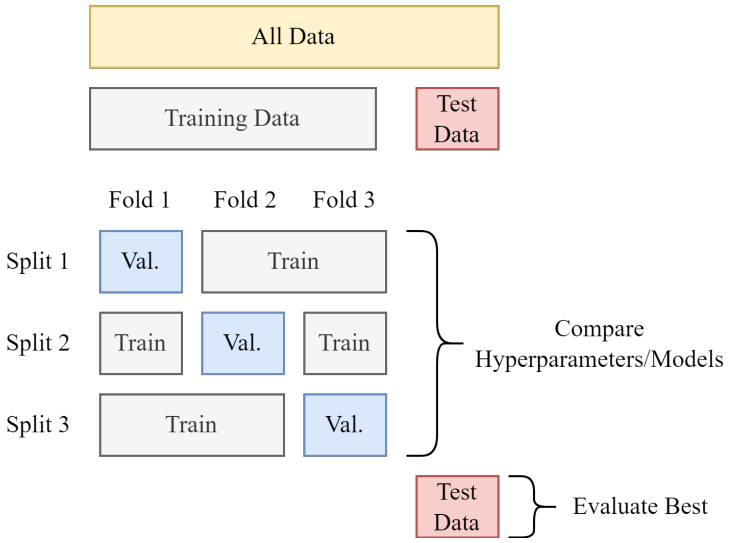



Figure 2.3: K-Fold Cross-Validation

## 2.3 Optimization Theory

Optimization theory is a field of mathematics devoted to solving optimization problems. An optimization problem can be defined as a function with a set of inputs and outputs, where the goal is to find the global minimum or maximum of the given function. Machine learning and almost every other sub-field of AI is built off solving different types of optimization problems, whether it be maximizing the distance between a set of points in an embedding, or minimizing the cross-entropy loss for an ANN in a classification type scenario. The ability to minimize these error functions quickly and efficiently has allowed for new breakthroughs in Machine Learning and AI.

Optima, otherwise known as critical-points, or extrema, are points of a function where the first derivative function value is equal to zero, or does not exist. There are four types of critical-points: weak, strong, global, and saddle points. The goal for optimization algorithms is to find the global minimum or maximum

critical point, which is defined to be the critical point whose function value is either the global maximum or global minimum across all critical points. However, it is common for optimization algorithms to become trapped in local weak extrema, which represent suboptimal solutions. Please see Figure 2.4 on how critical points are depicted in a univariate function.

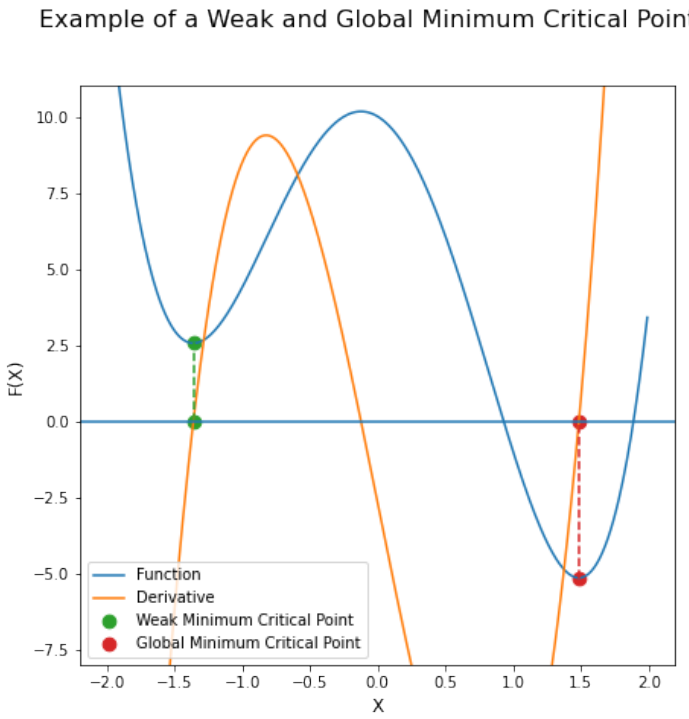



Figure 2.4: Example Function with Derivative and Minimum Critical Points

Many different types of optimization algorithms have been proposed in order to solve such optimization problems. They can be divided into three main categories: classical numerical methods, guided random search techniques, and local greedy searches. Classical numerical methods encompass methods that utilize knowledge concerning the gradient or Hessian matrix, the first and second derivative of the function. Such algorithms include Newton's method and gradient descent. Guided random search techniques encompass algorithms that do

not utilize knowledge of the gradient or hessian matrix, such as evolutionary algorithms, leap frog, particle swarm optimization, or simulated annealing. Lastly, local greedy searches are single population algorithms that also do not require derivative knowledge, but search locally about a point by taking small steps in different directions, such as hill climber.

Two of the three categories are commonly utilized throughout this work, classical numerical methods and guided random search techniques. Because both methodologies are vital for understanding optimization, both will be described in detail.

### 2.3.1 Classical Numerical Methods

Classical numerical methods can be described as a set of algorithms that have been developed to find global critical points by knowledge of the gradient or Hessian matrix. In machine learning models, the objective function to optimize is a loss function, typically MSE for regression or cross-entropy for classification. For both loss functions, optimization refers to finding the set of weights or parameters that will minimize the error, i.e. find the global minimum critical point. The inputs to these loss functions would be the weights, the set of trainable parameters, and the output would be the loss function value across the data instances.

Classical numerical methods are iterative algorithms that begin with an initial guess and iteratively update that guess in order to be closer to the global minimum critical point. The initial guess represents the initial input to the loss function, some type of random initialization of the weights and trainable parameters. Most iterative numerical algorithms follow equation 2.3 for updating

trainable parameters:

$$w_{i+1} = w_i - \alpha d_i \tag{2.3}$$

where $w_i$ are the weights of the current iteration, $\alpha$ is the learning rate, and $d_i$ represents the search direction of the current iteration. The search direction signifies which direction the weights are to step towards, while the learning rate scales the direction in order to control the magnitude of the step. The two most commonly utilized numerical algorithms for optimization in machine learning is gradient descent and stochastic gradient descent (SGD). Gradient descent uses the sum of the gradients evaluated at the previous weight across all data points as the search direction, as seen in equation 2.4, where $\nabla$ is the partial derivative of the loss function with respect to the weights, evaluated at the current weight $w_i$.

$$w_{i+1} = w_i - \alpha \nabla F(w)|_{w=w_i} \tag{2.4}$$

The difference between gradient descent and SGD is that gradient descent creates the step direction by summing the gradients for all points, while SGD creates the step direction by calculating the gradient at a single point. In this way, SGD would update the weights a total of $n$ times before the number of points evaluated would equal that of gradient descent. The benefit of utilizing SGD over standard gradient descent is that it is computationally cheaper to update the weights, as the gradient is only evaluated at a single point; however, it will take longer to converge as the calculated gradient is biased towards a single point instead of the population. In order to overcome this problem, an extension of SGD known as mini-batch SGD, partitions the training dataset into batches for calculating the gradient. For the rest of this thesis, when SGD is

mentioned, mini-batch SGD is assumed. Now, SGD updates the weights through the direction from the mean of the gradients across the batch.

$$w_{i+1} = w_i - \alpha \frac{1}{n} \nabla \sum_{j}^{n} F(w)|_{w=w_i} \tag{2.5}$$

This can be seen in Equation 2.5, as the step direction is replaced with the average of the gradients across the batch, where $i$ is the $i^{th}$ weight, $n$ is the total size of the mini batch, and $j$ is the $j^{th}$ sample from the mini batch. Now, the only hyper-parameter to tune would be the batch size. Large batch sizes better represent the population but require longer to update the weights, while smaller batch sizes update the weights faster at the cost of being biased towards the batch.

## 2.4 Evolutionary Computation

Evolutionary Computation is a sub field of CI where the goal is to mimic biological behavior in order to create intelligence. The two most common fields of EC are Genetic Algorithms (GA) and Particle Swarm Optimization (PSO). Both GA and PSO algorithms are population-based guided random search algorithms for solving optimization functions. In contrast to classical numerical methods, guided random search algorithms do not utilize knowledge about the function, like the gradient or Hessian matrix. As a result, guided random search algorithms can be applied to non-differentiable functions. First, GAs will be introduced along with some of its variants. Next, PSO will be introduced in two of its most popular forms.

### 2.4.1 Genetic Algorithms

Genetic algorithms seek to mimic the process of evolution in order to optimize optimization functions. From its biological inspiration, evolution can be described as a process by which individuals become more fit through three main mechanisms: natural selection, differential reproduction, and mutation. Natural selection occurs when individuals, that are poorly adapted for their environment, have higher probabilities of dying and not reproducing. As a result, individuals that are better adapted to their environment than others have higher probabilities of surviving and reproducing. Differential reproduction occurs subsequently from natural selection as only those fit enough to survive their environment are those able to reproduce. Because only those fit enough to survive reproduce, their offspring will share the genetic material of their parents, which in tern will produce offspring that are inherently more fit than their predecessors. Mutations occur either randomly, by chance of their genome; or, are guided through adaptation from selective breeding. GAs leverage these three processes for optimizing optimization functions.

Translating these processes to numerical mechanisms has posed an interesting task for AI practitioners. However, a general outline has been established. First, the fitness-function itself needs to be chosen. The fitness-function is a function that maps the chromosome representation of the individual to a scalar fitness value based on the optimization problem. Second, the representation of individuals within the genetic algorithm needs to be created. Third, the selection operators for choosing which individuals reproduce needs to be engineered. Fourthly, the reproduction operators detailing how the offspring will be created from the parents needs to be established. Lastly, the survival operators for se-

lecting which offspring and/or parents are to survive needs to be decided. The general overview of evolutionary algorithms can be seen in Figure 2.5.

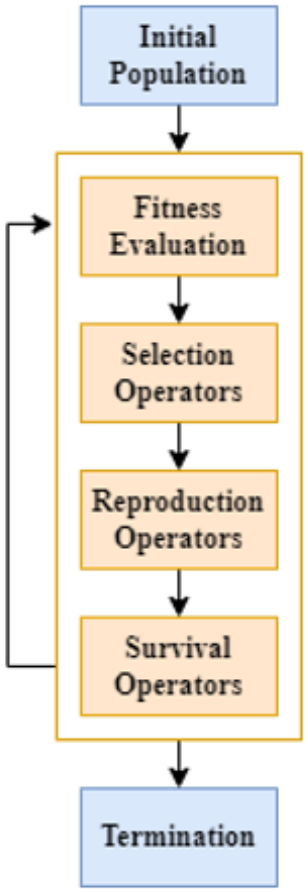



Figure 2.5: Flowchart of a Basic Genetic Algorithm

First, the initial population is created. The exact implementation of this initialization depends upon the practitioners goals of exploration and exploitation, a topic that will be discussed later on. After this initialization, the algorithm enters into a loop, where each iteration of the loop is known as a generation. Each individual in the population is evaluated by the optimization function to obtain its fitness score; assuming minimization, individuals with lower fitness scores indicate better fitness. Next, the selection operators are applied to select which

individuals from the population are to reproduce offspring. After selection, the offspring are created from applying the reproduction operators between the sets of parents. Finally, the survival operators are applied to the offspring and parents to select which individuals will survive on to the next generation. Termination occurs later on after certain criteria have been met.

### 2.4.2 Representation

From its biological inspiration, cells can be regarded as the basic building block of organisms. Each cell is built off different strands of chromosomes, where each chromosome is composed of DNA containing many different units of genes, which are regarded to be the fundamental components of heredity. The entire composition of genes is known as the genome. The genotype is regarded as a set of genes from an individuals genome. On the other hand, the phenotype, refers to how the genotype of an individual is expressed in its environment.

When constructing the fitness function, $f$, can be described more formally as a function that takes in the phenotype representation of the chromosome, $\Gamma$, and maps it into a scalar value from the Real domain, $\mathbb{R}$:

$$f : \Gamma \rightarrow \mathbb{R} \tag{2.6}$$

Typically, genetic algorithms represent the individual by its chromosome, an $n_x$-dimensional vector with either floating point or binary numbers. As a result, each variable value from the $n_x$-dimensional vector represents a gene. On the other hand, the phenotype is completely dependent on the optimization function. For multivariate optimization functions that originate from the Real domain, the phenotype of an individual is regarded as a point in space. For other types of

optimization functions, the genotype and phenotype will change based upon the problem at hand. For example, as will be described later, the phenotype of an individual can be a convolutional neural network, whereas its genotype is the encoding of the network in a fixed length multivariate domain.

An intermediate function is needed in order to convert the chromosome of an individual to its phenotype to obtain its fitness score from the fitness function. Altogether, the fitness function can be described as a function $f$, that accepts the $n_x$-dimensional vector of the chromosome $S_c$, encodes it into its phenotype representation using the function $\Phi$, and passes it through the objective function to obtain its scalar fitness value:

$$f : S_c \xrightarrow{\Phi} \Gamma \rightarrow \mathbb{R} \tag{2.7}$$

### 2.4.3 Exploration vs. Exploitation

Exploration is defined to be the process of searching the domain space of an optimization function in order to find newer and better solutions. On the other hand, exploitation, otherwise referred to as convergence, refers to the process of refining the current best solution in order to refine the most promising areas of the domain space. The exploration vs. exploitation tradeoff plagues many areas of optimization algorithms, as practitioners want to both explore for newer solutions while also refining the current best. In GAs, exploration and exploitation are handled by their selection, reproduction, and survival operators.

Selective-pressure refers to how long it would take for a population-based search algorithm to create a totally uniform population. In this way, algorithms with high selective pressure decrease diversity, i.e., exploration, in favor of ex-

ploitation; while algorithms with low selective pressure increase diversity while having poor exploitation.

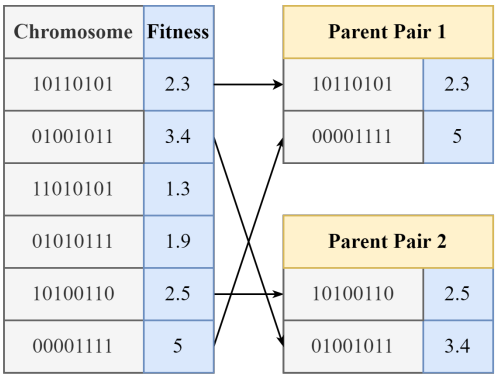
The purest form of an exploratory algorithm would simply be a pure random search by sampling from the domain space. In this way, the search space is explored much greater than other types of search algorithms as there is no refinement of the current best individuals. On the other end, the purest form of an exploitative algorithm would be a single population hill climber whose step-size is extremely small relative to the domain space. In this way, the entire search space of the optimization function is not explored; however, the current solution is refined to a degree such that it finds a minimum or maximum critical point, either global or local.

### 2.4.4 Selection Operators

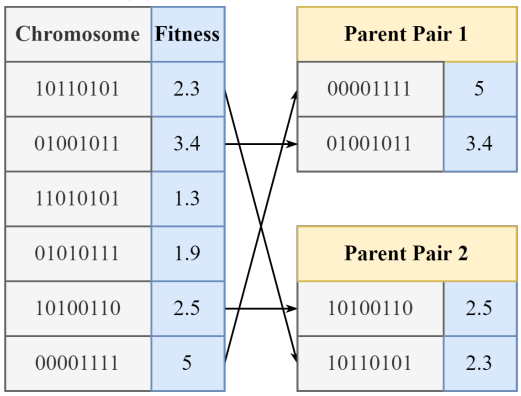
Once the optimization function, representation, and initial population have been established, each individual in the current population is evaluated to obtain its fitness score, i.e. its function value. From this list of fitness scores, the selection operators can be applied to select which individuals will reproduce with one other. The overall goal of selection operators is to mimic selective breeding by pairing up individuals that are more fit in order to create fitter offspring. There are four popular types of selection: random, proportional, tournament, and elitism. The number of parents varies between implementation, but standard GAs only utilize two parents, while paring up parents with, or without, replacement. Random selection is implemented by randomly pairing up individuals within the population with no regard to their fitness scores. By doing so, this selection procedure favors exploration over exploitation. On the other end of the spectrum, elitism

is implemented by only pairing up individuals with the best fitness scores, favoring exploitation over exploration. The middle-ground between the two comprises of both proportional and tournament selection. Proportional selection creates a probability distribution from the normalized fitness values of the individuals and randomly selects parents proportional to their normalized fitness value. Both exploration and exploitation are maintained as the individuals fitness value is normalized to a probability. In this way, individuals with better fitness values have larger probabilities of being selected while also allowing for poorer individuals to reproduce for the sake of exploration. Tournament selection works by randomly selecting groups, i.e. tournaments, of individuals where only the best individual from the group is selected. The tournament size controls the exploration vs. exploitation trade-off, as a tournament size equal to the population size is equivalent to selecting the best individual from the population, while a tournament size of one is equivalent to random sampling.

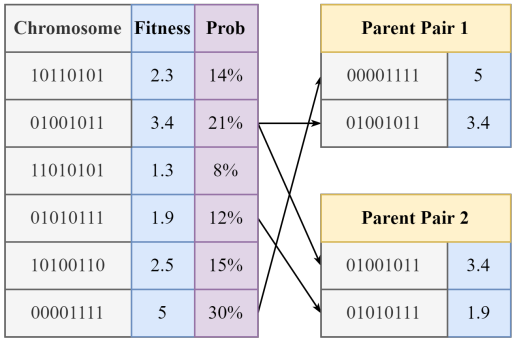
One can see examples of these in Figure 2.6. Note, in Figure 2.6, the objective is to maximize the fitness scores. In each subplot of Figure 2.6 there are six total individuals along with their respective chromosome and fitness value. Subplot 2.6*a* show random selection for two parent pairs, where the parents are chosen uniformly random from the population. Subplot 2.6*b* showcases elitism by sorting the individuals by their fitness value, and then pairing up individuals from that sorted order. Subplot 2.6*c* showcases proportional selection by calculating the probabilities of the individual being selected by dividing its fitness score by the sum of fitness scores. Parents are then sampled from these probabilities. Subplot 2.6*d* showcases tournament selection where the parents are selected by the winners of independent tournaments.



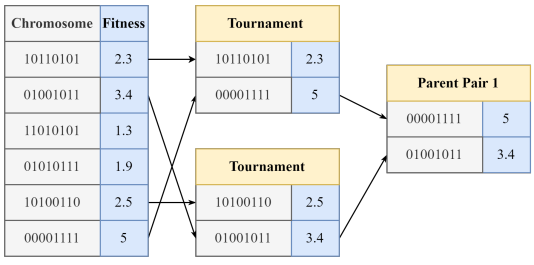

(a) Random Selection




(b) Elitism Selection




(c) Proportional Selection




(d) Tournament Selection

Figure 2.6: Selection Operators

### 2.4.5 Reproduction Operators

The goal of the reproduction operators is to utilize the existing genetic material of the parents, while also introducing new material, in order to further explore the domain space. Exploration is performed by two main operations, crossover and mutation. Crossover is utilized in order to globally search throughout the domain space, while mutation is used for local search. Crossover is implemented by selecting and assigning genes from the parents chromosome to the offspring, while mutation is implemented by randomly changing one or more of the genes. When the chromosome is continuous and fixed, crossover can be applied in two different fashions. First, the mean of the parents chromosomes can obtain to act as the crossover between the parents. Because the mean of the parents is deterministic, only one offspring will be produced. In order to create multiple offspring while utilizing mean crossover, the offspring can be created as a linear combination of the parents where the coefficient is sampled normally. This can be depicted in Equation 2.8, where $\tilde{x}$ represents the child, $x_1$ refers to the first parent, $x_2$ to the second parent, $i$ to the $i^{th}$ child, $j$ to the $j^{th}$ gene/variable, and $\gamma$ is the random variable sampled from the Gaussian Normal distribution with mean 0.5 and standard deviation 0.15.

$$\begin{gathered} \tilde{x}_{ij} = (1-\gamma)x_{1j} + \gamma x_{2j} \\ \text{Where } \gamma \sim N(0.5, 0.15) \end{gathered} \tag{2.8}$$

Second, point crossover can be applied. Point crossover is implemented by selecting one or more points in the genotype and simply swapping, i.e. crossing over, the sections of genes that fall between the two points. This process can be extended to uniform point crossover where all genes are inherited randomly

from the parents. Examples of this can be seen in Figure 2.7. In part $a$, the two parents and their chromosomes are depicted. In part $b$ there is mean crossover, in this instance calculated by taking the geometic mean between the parents. In part $c$, the seventh bit from the left is selected to be the point crossover where the genes of the parents are swapped to create two children. In part $d$, one point crossover is extended to multi-point crossover where the fourth and seventh bits are selected as crossover points. Lastly, part $e$ showcases uniform point crossover where genes are uniformly randomly swapped between parents.

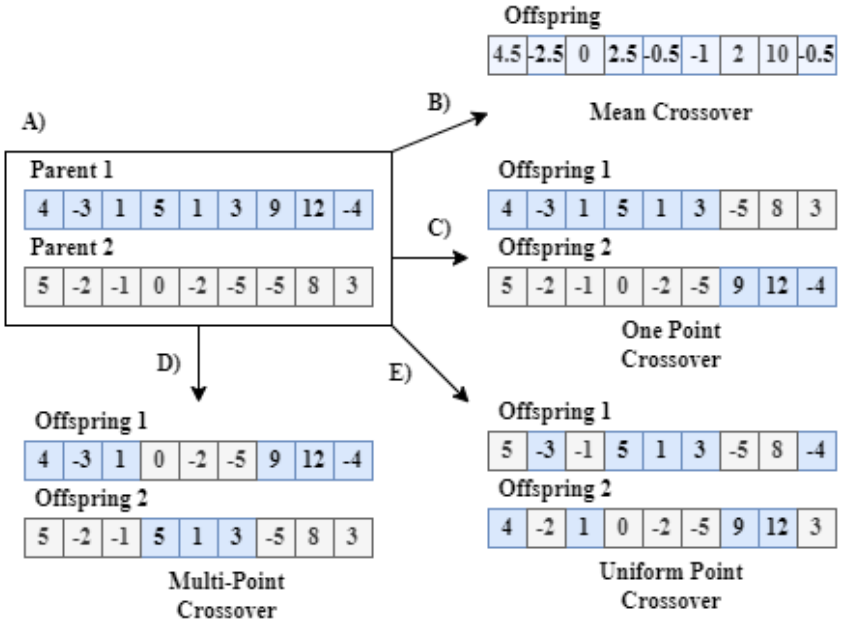



Figure 2.7: Point Crossover

Crossover globally explores the domain space as it can hop across the fitness landscape; however, its only flaw is that it can only explore the genotype space that is already known, as it relies upon the variable values already existing within the population, unless mean crossover is utilized with a randomly sampled coefficient. In order to explore new areas, mutation is incorporated to introduce new

genetic material by slightly changing gene values. Mutation can be implemented by either randomly changing a single, or multiple, variable values by adding a uniformly random number within a given set of bounds, or by adding a mutation vector of small uniformly random numbers within a given set of bounds to the entire chromosome. The first option can be depicted in Equation 2.9, where the $j^{th}$ gene/variable of the child $\tilde{x}$ is changed by adding a small random number sampled from the uniform distribution with respect to the bounds of that variable. The latter version of mutation by adding a whole mutation vector can be depicted by Equation 2.10, where the child is changed by adding a vector samples from the uniform distribution with respect to the bounds of the entire chromosome.

$$\tilde{x}_j = \tilde{x}_j + U(-x_j^{bound}, x_j^{bound}) \tag{2.9}$$

$$\tilde{x} = \tilde{x} + U(-x^{bound}, x^{bound}) \tag{2.10}$$

Together, crossover and mutation are extremely powerful mechanisms for both searching globally and locally. In most genetic algorithms, performing crossover and/or mutation to every offspring is not wanted, but instead are performed probabilistically. In this way, if the offspring always replace the parents, then the information of the parents is not lost from the current generation to the next, as crossover and mutation are only performed probabilistically. In addition, by increasing the probabilities of crossover and mutation, the algorithm will favor exploration as the original genetic material of the parents will be lost as the offspring would have a higher probability of altering their makeup.

### 2.4.6 Survival Operators

After the reproduction operators have been applied to the parents to obtain the offspring, the survival operators are then applied to select which individuals from the population will survive on to the next generation. The goal of the survival operators is to mimic natural selection. The creation of the new population can be performed in a few different fashions. One possibility is that the offspring immediately replace the parents. A second possibility is that the offspring and parents are pooled together, from which the survival operators will be applied to the joined combination of parents and offspring. Assuming the first option, the survival operator would simply be taking the offspring in favor over the parents. However, if the number of offspring is greater than the number of parents, then any of the selection operators discussed earlier can be applied for survival. On the other hand, if the survival operators are to be applied between the parents and offspring, then competition for survival can be either applied at the parent or population level. Parent level competition entails only applying the survival operators between the immediate parents and their immediate offspring. Parent level competition can be seen in Figure 2.8, which showcases two parent offspring pairs, where the winners by elitism within the offspring parent pairs are combined to form the next population. Competition between the entire population entails applying the survival operators between the overall pooled offspring and parents. The standard survival operators applied are the same as for selection. Global level competition can be seen in Figure 2.9, which utilizes the same parent offspring pairs from Figure 2.8 except pools the parents and offspring together. From these two pools, the winners are selected through elitism to obtain the next population. Parent level competition slows down exploitation while global level competition

speeds up exploitation as the fitness values of all parents and offspring are known and can be utilized during the selection process.

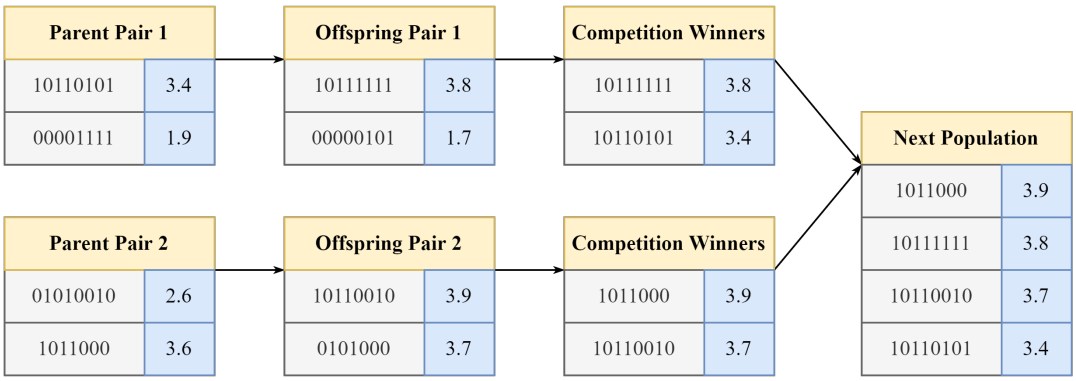



Figure 2.8: Parent Competition

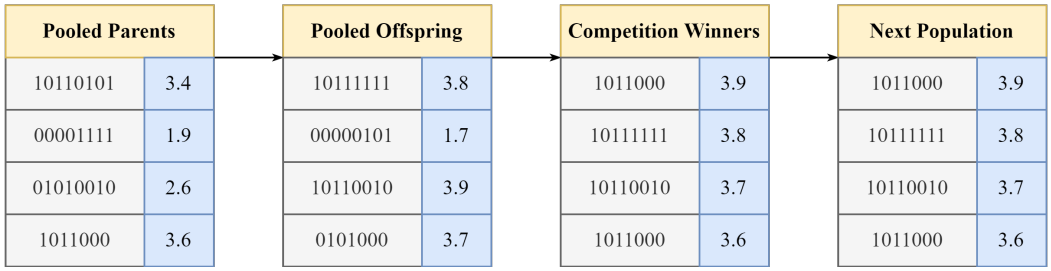



Figure 2.9: Global Competition

### 2.4.7 Termination

Termination of the genetic algorithm can occur in a few different ways. The simplest approach would be to stop when the set number of generations has been reached. Alternatively, termination can occur when the mean fitness values has stalled for a set number of generations. Early stopping utilizing the mean fitness instead of the best fitness is necessary as the best fitness can be static for a long period of time while the mean fitness can still be obtaining better values as the selection and survival operators will slowly start to converge to the best.

### 2.4.8 Control Parameters

Genetic algorithms contain many hyper-parameters that need tuning in order to be successful. Specifically, initial population size, generation size, number of generations, probabilities of crossover and mutation, and max mutation bounds. The probabilities of crossover and mutation refer to the probabilities of performing crossover or mutation on a given offspring. The max mutation bounds refers to a set of bounds that contain the max values for each gene that the offspring is able to be mutated. These hyper-parameters directly control the exploration vs. exploitation trade-off. Large generation sizes allow for more exploration as the algorithm is capable of holding more diversity within each generation at the cost of poorer convergence. The probabilities of crossover and mutation directly control exploration and exploitation as higher probabilities favor exploration while lower probabilities favor exploitation. The max mutation bounds also contribute to exploration and exploitation as lower max mutation bounds favor exploitation as the chromosome is only slightly changed, while larger max mutation bounds favor exploration as the chromosome is greatly changed from the larger perturbation.

These three hyper-parameters: probabilities of crossover, mutation, and max mutation bounds, can either be static, dynamic, or self-adaptive. Static hyper-parameters stay the same throughout the entire lifetime of the algorithm, whereas dynamic parameters typically decrease from large values to small values over the course of the algorithms lifetime. An example of a dynamic schedule is logistic decay, which logistically decreases the parameters value with each subsequent generation. Lastly, self-adaptive parameters adapt by either increasing or decreasing depending upon certain statistics of the current algorithm's execution. For example, a simple self-adaptive schedule increases the probabilities of crossover

and mutation if the mean fitness value approaches the best fitness, indicating too much convergence, in order to encourage more exploration, while decreasing the probabilities if the gap between the mean and best fitness becomes too large, indicating too much exploration, in order to encourage more exploitation.

The selection of these hyper-parameters can become very problematic for practitioners as they directly control the success of the algorithm. As a result, this higher level, abstract, view of the hyper-parameters can turn into an optimization function itself. In circumstances when the objective function is extremely computationally expensive to evaluate, it can be inefficient and futile to try to optimize these higher level parameters that control the run, instead relying on the initialization of the practitioner. Later on, during the explanation of the genetic algorithm utilized in this work, a simple work-around will be provided in order to escape from this dilemma.

### 2.4.9 The Basic Genetic Algorithm

Now that the necessary background information has been discussed over the basics of genetic algorithms, a simple procedure can describe the algorithm. The basic genetic algorithm can be detailed in Algorithm 1. The algorithm takes in the fitness function $F$, along with the selection, reproduction, and survival operators. The algorithm first initializes the population by randomly sampling from the domain space, and then obtains their respective fitness. The algorithm enters into a for loop where the parents are chosen from the current population and their associated fitness values utilizing the selection operators. After the parents are selected, the offspring are created utilizing the reproduction operators, and then their associated fitness values are obtained. Lastly, the population of the

next generation and its fitness values are calculated through the survival operators. This loop continues for a total of *max_gen* iterations, after which the best individual is returned as the final solution.

**Algorithm 1** Basic Genetic Algorithm

**Input:** Fitness Function, $F : S_c \xrightarrow{\Phi} \Gamma \rightarrow \mathbb{R}$; max_gen; Selection, Reproduction, and Survival Operators

```
population = random_initialization()
fitness = fitness_function(population)
for i until max_gen do
    parents = selection_operators(population, fitness)
    offspring = reproduction_operators(parents)
    fitness_offspring = fitness_function(offspring)
    population, fitness = survival_operators(parents, fitness,
    offspring, fitness_offspring)

end
```

**Return** Best Individual from Population

## 2.5 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is another branch of CI for optimizing objective functions, but seeks to do so by mimicking the behavior of swarms. From its biological inspiration, the collective behavior of animals or insects in swarms is typically performed in order to find resources, fend off predators, or migrate to breeding or resting grounds. The unpredictable ability for swarms of birds to fly independent from each other, yet maintain a social behavior of flying in formation with the flock intrigued many early researchers. It can observed that birds maintain both cognitive and social experience. The birds fly where their experience and intuition drive them, which represents their cognitive experience, while at the same time maintaining the structure and direction of the flock, their

social experience. The basic flowchart for PSO algorithms can be seen in Figure 2.10.

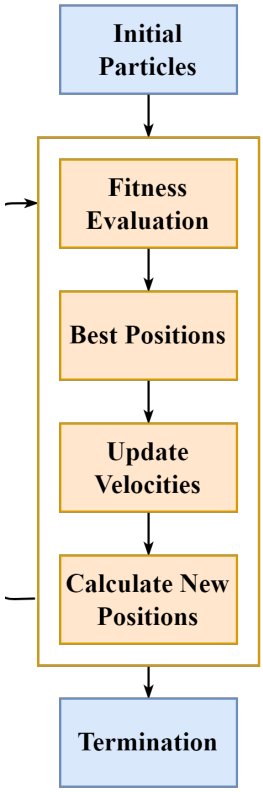



Figure 2.10: Basic PSO Flowchart

### 2.5.1 Global Best and Local Best

Translating the behaviors of swarms to an algorithm for optimization has been extremely successful. Like GAs, PSO algorithms are population-based algorithms where each individual in the population represents a particle. Each particle maintains a current position and keeps track of three components: current velocity and cognitive and social components. Like GAs, particles are represented by their genotype and phenotype. Their genotype would be the multidimensional

vector of variable values, while their phenotype would be the implementation of their genotype in their environment.

Each particle in a PSO algorithm keeps track of their own independent velocity and position vectors. Particles in PSO algorithms are updated by adding their current position to their new velocity component, as seen in equation 2.11.

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{2.11}$$

The initial positions of the particles correspond to the initialization of the initial population for GAs. With each iteration of the PSO algorithm, the new velocity vector is calculated by the linear combination of the particles current velocity, cognitive, and social components. This can be seen in equation 2.12.

$$v_i(t+1) = wv_i(t) + c_1r_1(y_i(t) - x_i(t)) + c_2r_2(\hat{y}_i(t) - x_i(t)) \tag{2.12}$$

In equation 2.12, $v_i(t)$ is the particles current velocity, $(y_i(t) - x_i(t))$ represents the cognitive component, and $(\hat{y}_i(t) - x_i(t))$ refers to the social component. In both the the cognitive and social components, $x_i(t)$ refers to the particles current position at time $t$. The cognitive component is the difference vector between the particles personal best position, $y_i(t)$, and the particles current position. Each particle keeps track of its history of positions along with their respective fitness scores. The personal best position is the position in the history of positions for a particle where its fitness score is the best, where best is dependent upon maximization or minimization. The social component is the difference vector between the best position, $\hat{y}_i(t)$, and the particles current position. The best position can either be defined to be the best position globally, across all particles, or locally, only from some number of surrounding neighbors. If the best position

is the best position globally across all particles, the PSO algorithm is referred to as gbest. On the other hand, lbest is another type of PSO algorithm where the best position is the best position from a group of neighboring particles. The exact implementation of this neighborhood will not be detailed as lbest is never used in this research work.

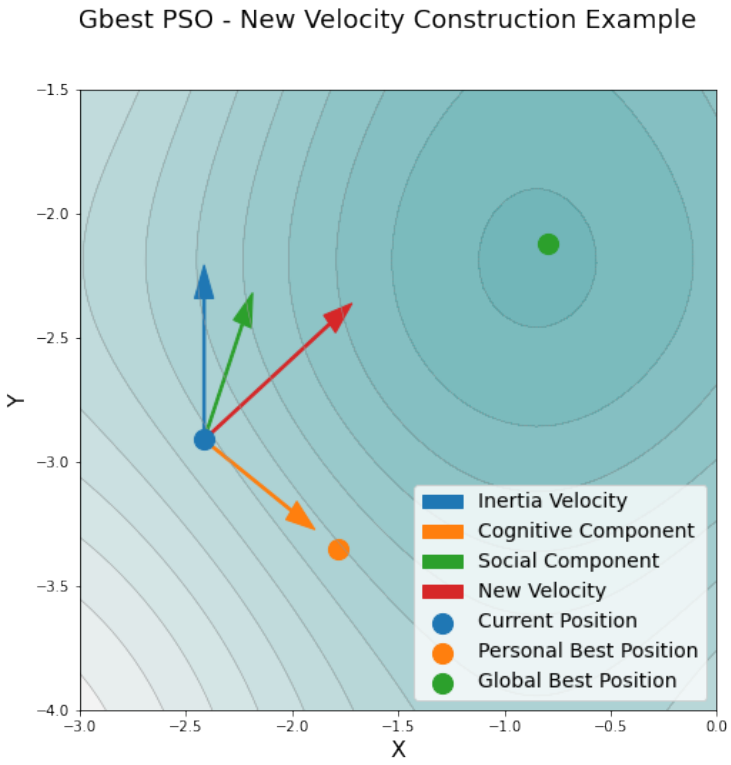



Figure 2.11: Geometric Representation of gbest Components

The geometric representation of the components from the velocity vector can be seen in Figure 2.11. In Figure 2.11, there are three points, the current particles position in blue, its personal best position in orange, and the global best position in green. After creating the difference vectors, the green arrow represents the social component multiplied by a small random number and the $c_2$ coefficient, while the orange arrow represents the cognitive component, multiplied by a small

random number and the $c_1$ coefficient. Lastly, the particles current velocity is depicted by the blue arrow, representing its current inertia, multiplied by the $w$ coefficient.

Because gbest utilizes the global best position across all particles, it achieves very good convergence; however, it can lack exploration as it can commonly become stuck in local optima. The complete gbest PSO algorithm is given in Algorithm 2:

**Algorithm 2** gbest PSO Algorithm

**Input:** Fitness Function, $F : S_c \xrightarrow{\Phi} \Gamma \to \mathbb{R}$; max_iter; $w, c_1, c_2$ coefficients

```
particles = random_initialization()
velocities = random_initialization()
for i until max_iter do
    fitness = fitness_function(particles)
    gbest_position = best_particle()          ▷ # Best global Fitness
    for each particle do
        update velocity using equation 2.12
        update position using equation 2.11
    end
end
```

**Return** Best Particle from Swarm

On the other hand, lbest has the ability to escape from local optima because its current best position is only defined to be the best position from amongst the particles set of neighbors. The complete lbest PSO algorithm is given in Algorithm 3:

**Algorithm 3** lbest PSO Algorithm

**Input:** Fitness Function, $F : S_c \xrightarrow{\Phi} \Gamma \rightarrow \mathbb{R}$; max_iter; $w, c_1, c_2$ coefficients

```
particles = random_initialization()
velocities = random_initialization()
for i until max_iter do
    fitness = fitness_function(particles)
    lbest_position = best_particle()                ▷ # Best local Fitness
    for each particle do
        update velocity using equation 2.12
        update position using equation 2.11
    end
end
```

**Return** Best Particle from Swarm

### 2.5.2 Control Parameters

Like GAs, PSO algorithms have their own set of hyper-parameters that need fine tuning. Besides the initial population size, swarm size, and number of iterations; PSO algorithms have to tune three extremely vital coefficients that control the success of the algorithm: $w$, $c_1$, $c2$. The $w$ coefficient is applied to the velocity vector during its update. It is commonly referred to as the inertia coefficient. This parameter controls how much the previous velocity will impact the construction of the current velocity. Large inertia values lead to exploration while small values lead to exploitation. The parameters $c_1$ and $c_2$ control the influence of the cognitive and social components respectively. Having $c_2$ set to 0 makes the algorithm a version of hill climber that utilizes only its own experience. Having $c_1$ set to 0 makes the algorithm a purely greedy algorithm that converges to only the current best solution. It is typical that $c_1$ and $c_2$ need to balance each other, as increasing one will intuitively decrease the other. In this way, $w$, $c_1$, and $c_2$ control the exploration vs. exploitation trade off. These three hyper-

parameters are vital for achieving success. Like section 2.4.8 over parameters in GAs, the same methods for updating these hyper-parameters can be applied here to PSO parameters. These three coefficients can be either static, dynamic, or self-adaptive. Dynamic implementation starts with large $w$ and $c_1$ values to encourage exploration, but slowly decrease them while simultaneously increasing $c_2$ in order to achieve convergence. Self-adaptive mechanisms update these three coefficients depending upon the statistics from the current iteration of the algorithm, as described in section 2.4.8.

## 2.6 Artificial Neural Networks

### 2.6.1 Introduction

On the other end of the spectrum in CI, Artificial Neural Networks (ANNs) were first created with the goal of modeling artificial intelligence by trying to model biological neural networks found in the brain. ANNs have been shown to be successful at learning representations for classification, function approximation, and pattern recognition. ANNs first evolved from the appearance of the perception, which was later scaled to deep feed-forward networks containing multiple hidden units and layers. With performance issues on image recognition tasks rising, convolutional neural networks (CNNs) were created in order to handle the new data types. CNNs were shown to be extremely successful and efficient, opening up a whole new field of research for researchers to dive into. Over the past couple of years, many different types of CNN architectures and models have been created, namely, AlexNet, VGGNet, ResNet, and Inception. Each model has their own unique set of additions and contributions that have helped form the basis

of advanced CNN research today. With more advanced model architectures, the problem of over-fitting became more prevalent to a point that researchers started to dive into reducing such occurrences through the addition of new layers and augmentation techniques. ANNs and CNNs are known for being extremely large, containing up to tens of millions of parameters, creating a very difficult optimization problem. As a result, advanced optimizers have been explored in order to find better weight solutions faster and more efficiently.

### 2.6.2 The Artificial Neuron

As the name implies, the artificial neuron is built off the intuition of the biological neuron. Biological neurons each take in a set of inputs, referred to as synapses, apply some type activation function to the synapses, and then outputs a signal. From this inspiration, an artificial neuron is a function that maps a set of inputs to a non-linear output. The artificial neuron is capable of learning representations for two particular scenarios: classification and regression. The artificial neuron can be depicted in Figure 2.12, where $x$ represents an input, $w$ represents a learnable weight, $f$ represents a non-linear activation function, and $o$ represents the corresponding output.

In Figure 2.12, $net$, is calculated to be the weighted sum between the inputs and the weights, plus the bias term $b$, the intercept. This can be seen in Equation 2.13.

$$net = b + \sum_{i=1}^{n} x_i w_i \tag{2.13}$$

There have been many activation functions created in recent years, but the most foundational function first applied was the sigmoid function, depicted in

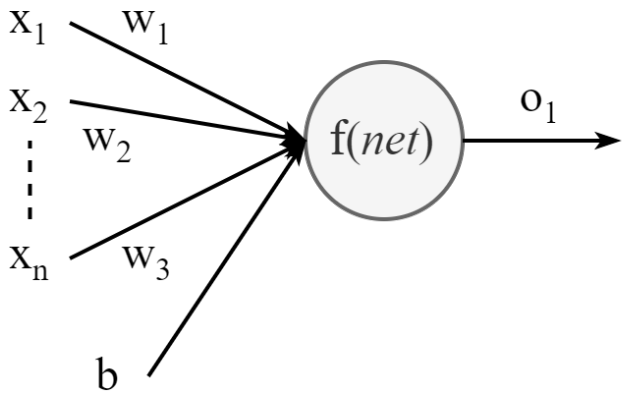



Figure 2.12: Perception

Equation 2.14, where $\lambda$ is a hyperparameter that is set by the user, typically set to 1.

$$f(net) = \frac{1}{1 + e^{-\lambda(net)}} \tag{2.14}$$

The goal for an activation function is to obtain non-linearity between the input and output, so that the weight parameters can be optimized to the point that the predicted output from the perception either matches or closely resembles the actual output. The sigmoid function outputs a value between $[0, 1]$; great for regression tasks where the actual output has been normalized to this scale, or for binary classification where values above 0.5 go to class 1, while values less than 0.5 go to class 0.

Artificial neurons are capable of learning through gradient descent, which updates the weight values through the gradient-descent algorithm depicted in Equation 2.4, re-iterated here in Equation 2.15 for ease. Assuming one utilizes the mean squared error function (MSE), given in Equation 2.16, the partial derivative

is given in Equation 2.17, where $\frac{\partial f}{\partial net}(x_i)$ is the partial derivative of the activation function with respect to the weights, evaluated at $x_i$.

$$w_{i+1} = w_i - \alpha \nabla F(w)|_{w=w_i} \tag{2.15}$$

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{2.16}$$

$$\nabla F(w)|_{w=w_i} = -2(\hat{y}_i - y_i)\frac{\partial f}{\partial net}(x_i) \tag{2.17}$$

### 2.6.3 Dense Neural Networks

Artificial neurons were extremely successful; however, they were incapable of learning most types of complex functions and non-linearly separable classification problems. Abstracting outward to biological neural networks, which are comprised of millions of artificial neurons, artificial neural networks are composed of many artificial neurons. The most basic artificial neural network is known as a feed-forward neural network, indicating that all of the connections between neurons goes forward throughout the network, none backward. Feed-forward networks compose of an input layer, an output layer, and any number of hidden layers composing of any number of hidden units, neurons. Hidden units refer to neurons that are not directly observed by either the input or output layers. Feed forward networks can be depicted as below in Figure 2.13, with an arbitrary number layers with each an arbitrary number of inputs and outputs

Figure 2.13 showcases an example of a feed forward neural network with an input layer, composed of the variable values from $X$, the first hidden layer,

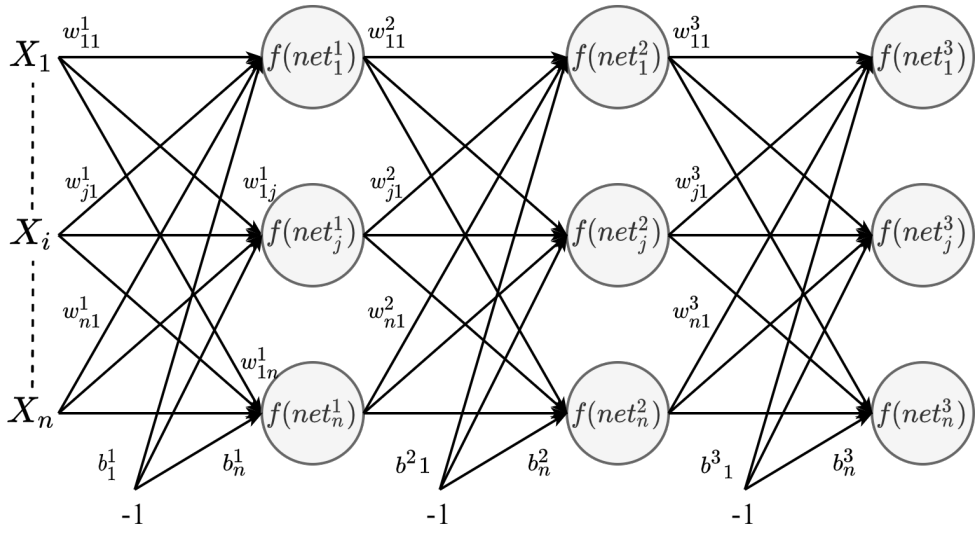



Figure 2.13: Example Feed-Forward Neural Network

composed of neurons from $f(net^1_1)$ to $f(net^1_n)$, a second hidden layer, composed of neurons from $f(net^2_1)$ to $f(net^2_n)$, and finally an output layer, composing of multiple output neurons, $f(net^3_1)$ to $f(net^3_n)$. Each layer has their own set of unique, independent, weights and biases. Due to the mathematical nature of the network, the weights and biases for each layer can be represented as matrices and vectors, $W^j$ and $B^j$, where $j$ is the hidden layer. As a result the *net* input for each layer now becomes the matrix multiplication of its input matrix and its current weight matrix plus the bias vector. This is shown in Equation 2.18, where $f(net)^{j-1}$ is the output from the previous $j^{th}$ layer.

$$net^j = f(net)^{j-1}W^j + B^j \tag{2.18}$$

Updating the weights of feed-forward neural networks is similar to the perception, as gradient descent or SGD can be utilized. However, because feed-forward neural networks utilize multiple hidden layers, where the output of each layer

is composed of inputs evaluated multiple times at different activation functions, it requires a methodology known as back propagation in order to calculate the partial derivative for the weights. Back propagation calculates the gradient with respect to the weights of layers using the chain rule of calculus. As a result, the gradients of the earlier layers of networks are calculated by a multiplicative chain of the gradients from the subsequent layers.

### 2.6.4 Convolutional Neural Networks

#### Introduction

Standard images are stored in memory as a three-dimensional matrix: one dimension for the height of the image, one for the width, and another for the color channel. Colored images contain three color channels: red, green, and blue; referred to as RGB. As an example, suppose a colored image is 128x128; it is stored in memory as a 128x128x3 array; in this way, an image is comprised of three 128x128 matrices, one for each RGB channel. Because standard neural networks require a single feature vector as input, multi-dimensional feature matrices cannot be applied. To overcome this issue, images can be flattened into a single vector of input. From the previous example, the 128x128x3 colored image can be flattened into a 49,152 element feature vector for input. Although this solution allows for images to be processed by neural networks, the flattening procedure loses important information concerning the images, such as the spatial and temporal representation. Spatial representation refers to the surrounding neighbors of a pixel while the temporal representation refers to the depth of the pixel in the RGB channels.

An extension of feed-forward neural networks was created to solve this prob-

lem by utilizing convolutions, giving the model the name convolutional neural network, also referred to as ConvNet. Early ConvNets composed of three simple layers, convolutional kernels, pooling layers, and dense layers. Convolutional kernels and pooling layers will be discussed shortly, while dense layers refer to the simple feed forward networks described earlier.

### Convolutional Kernel

Convolutional layers in neural networks compose of a kernel and a bias that are applied to multi-dimensional inputs. For image processing, the kernel composes of a three-dimensional array of learnable weights that are applied to the input. This three-dimensional array has the same format as an image: a height, width, and third dimension for the number of channels. The height and width of the kernel must always be equal to or smaller than height and width of the input; however, the third dimension for the number of channels can either be greater or lesser than that of the input. The kernel for standard image processing can be described as a hyper-cube that is applied to subsections of the image, that is then slid over to the next set of pixels for convolution.

The convolution operation works by simply creating a linear combination between its weights and the selected pixel values from the input selection. Figure 2.14 shows this operation spatially as the input is a 4x4 input and the kernel is 2x2. The convolution operates beings by first selecting a subsection equal to the kernel size, applies the linear combination between the subsection and kernel and utilizes the sum as the output pixel. The kernel is then slid over one pixel to obtain the next subsection. Figure 2.14 shows that the output dimension shrinks by 1 on all sides as only three 2x2 kernels can be applied to the input while only sliding one pixel at at time without falling off the image. Padding can be

applied to keep the output dimensions equal to the input. In practice, the input is typically a three dimensional array, by which a three dimensional kernel is applied to the input, as show in Figure 2.15.

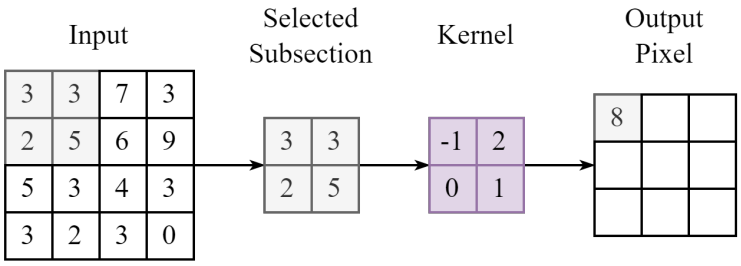



Figure 2.14: Spatial Convolution

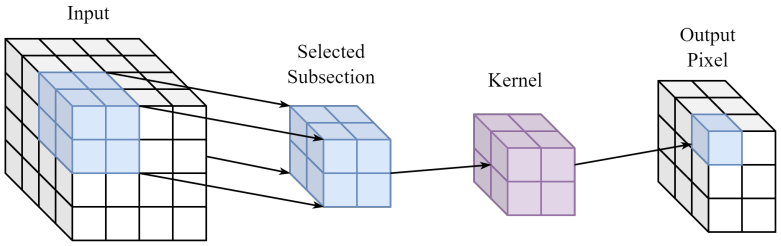



Figure 2.15: Spatial and Temporal Convolution

Convolutional layers comprise of four very important hyper-parameters: kernel size, depth, stride, and padding. Kernel size refers to what height and width the kernel should have, while depth refers to how many channels the kernel has. A 7x7 kernel refers to a kernel with both height and width of 7 pixels. Larger kernels heights and widths have more receptive range as they contain more pixels per operation. If the kernels size does not evenly fit into the input image dimensions, padding can be applied by adding a layer of zeros to the dimensions the image so that the output dimensions will match the input dimensions. Stride will be discussed in the Pooling Layer subsection. It is common in practice to

apply an activation function after a convolutional layer to obtain non-linearity, otherwise the convolution layer would simply be a weighted sum of pixels.

### Pooling Layer

The objective of pooling layers is to reduce the dimensionality of the input in order to both save computation and reduce noise. Pooling layers are invariant to small perturbations to the input, meaning that if the input is changed slightly the output from the pooling layers is still the same or extremely similar. There are two common pooling layers utilized in practice, max and average. Like convolution layers, pooling layers have kernels sizes only for their width and height, along with a stride. In this way, because pooling layers lack another dimension for output channels, the number of channels for the output is the same as the input. However, the kernels for both max and average pooling no longer consist of trainable weights but instead take either the maximum or average value for a selected subsection from the input.

Pooling layers are known for their reduction in computation by acting as a cheap alternative to convolution layers as they do not require trainable parameters. In addition, with the inclusion of a stride, pooling layers can reduce the dimensionality of the input along the width and height axes. Stride refers to how many pixels are slid across by the kernel. Up until now, all examples have utilized a stride of one, meaning that once the kernel has been applied to a subsection of the input, it is slide across by one pixel to obtain the new subsection of the input. With a stride of two, the input dimensions are cut in half. Strides can also be implemented in convolution layers in order to reduce dimensionality as well, but pooling layers are sometimes favored in practice because they are cheaper in computation and are less prone to overfitting. A convolution layer

with stride of two can be depicted in Figure 2.16, where each color corresponds to the subsection for the corresponding output pixel.

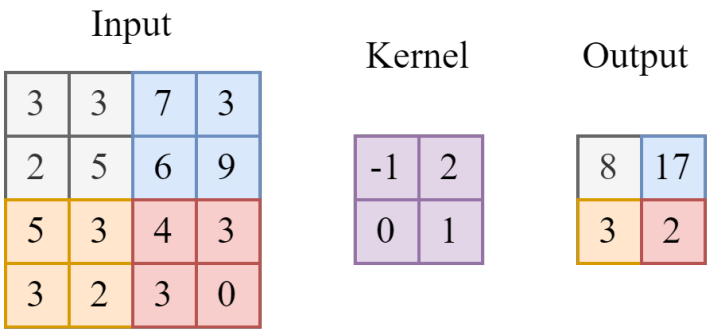



Figure 2.16: Spatial Convolution with stride of 2

**Simple ConvNet**

Now since the basic building blocks for a CNN architecture has been discussed, a basic ConvNet architecture can be constructed. Figure 2.17 showcases a simple ConvNet architecture where the input is a 224x224x3 image followed by three modules of max pooling, convolution, and ReLU activation. After these three modules, a max pooling layer is applied before the data is flattened into two subsequent dense layers before exiting to the softmax output layer. ReLU and softmax activation functions will be discussed shortly.

### 2.6.5 Activation Functions

The objective of an activation function is to apply a non-linear function to the input in order to obtain a non linear output. The sigmoid function was mentioned previously in Section 2.7.2, but now will be expanded upon. The sigmoid function is a monotonically increasing function ranging from 0 to 1. Because of the closed

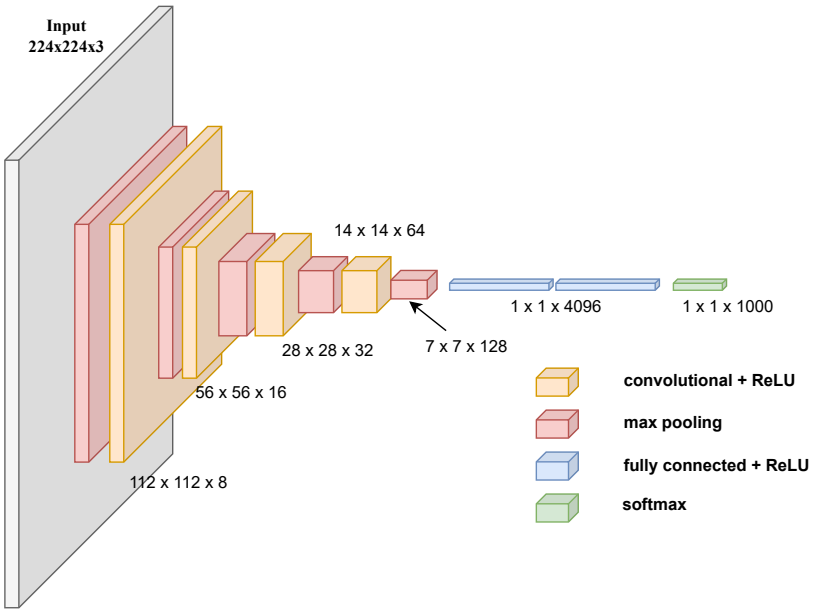



Figure 2.17: Basic ConvNet

bounds, the sigmoid function squeezes its input when projecting. As a result, neurons with negative sums will project extremely close to zero. As a result, these values near zero will force the neurons of the next layer to become less important as their weighted sum will be smaller, due to the near zero scaling from the previous neuron.

A closely related neighbor to the sigmoid function is the tanh function, given in equation 2.19.

$$f(net) = \frac{e^{net} - e^{-net}}{e^{net} + e^{-net}} \tag{2.19}$$

The tanh function is a monotonically increasing function that ranges from -1 to 1. Instead of turning off the neurons of the next layer when the previous sums are negative, the tanh function outputs a negative number, allowing for information to still carry on throughout the network. However, both the

sigmoid and tanh function greatly suffer from exploding or vanishing gradients when the network has many layers. Because larger networks contain many layers, the methodology of back propagation multiplies the gradients in chains of subsequent layers in order to obtain the gradient of current respective layer. When this chain of multiplications becomes too large, the gradient explodes, i.e., tends towards infinity; whereas when the chain of multiplications becomes too small, the gradient vanishes, i.e. tends towards zero. Exploding gradients often ruin a model as the model weights will tend towards infinity. On the other hand, vanishing gradients will not ruin a model but leave it stagnate as the magnitude of the gradient will not be large enough for the optimization algorithm to continue minimization.

Both the sigmoid and tanh function are extremely prone to both exploding and vanishing gradients as they heavily rely upon the initialization of their weights. When the initial weights are too large for a respective model, the back propagated error for larger networks will explode as the chain multiplication of a large error will tend toward infinity. On the other hand, because the derivative of the sigmoid and tanh function have relatively small max values, 0.25 for sigmoid and 1 for tanh, as can bee seen in Figure 2.18, the chain multiplication of extremely small gradients will compound towards zero.

These limitations of both sigmoid and tanh functions led researchers and practitioners alike to move from traditional activation functions to a family of activations known as rectified activation functions. ReLU, short for rectified linear activation unit, is an activation function, given in equation 2.20, that returns the max between 0 and $net$.

$$f(net) = \max(0, net) \tag{2.20}$$

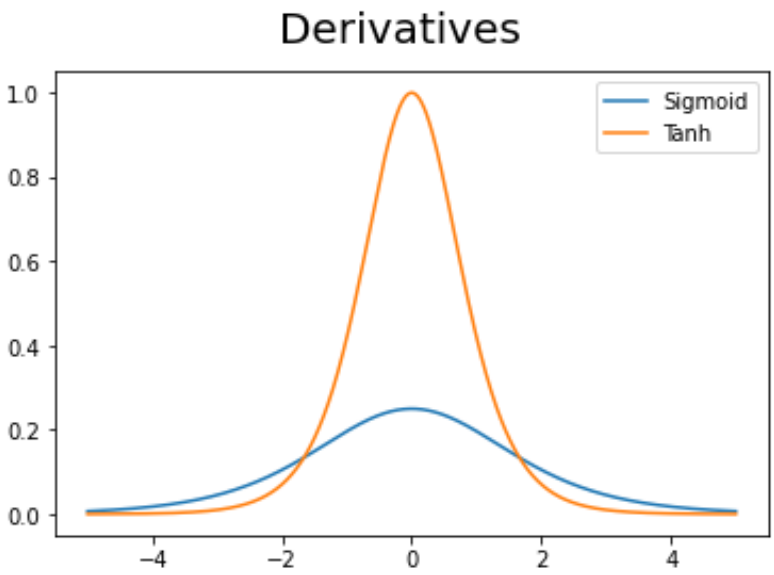



Figure 2.18: Derivatives for Sigmoid and TanH

As a result, negative input sums will turn off nodes of subsequent layers instead of letting a small amount of information pass. By turning off neurons, the input from one layer to a next becomes sparse. There are four primary advantages of sparsity given by the authors in Deep Sparse Rectifier Neural Networks [9]. The first is information disentanglement, which refers to how small changes in input lead to small changes in output. When the input to a layer is dense, the representation becomes entangled as small changes to the input will lead to larger changes in output. However, if the representation is sparse and robust, small input changes will lead to only small changes in the output. The second advantage is efficient variable-size representation. Because of sparsity, the number of active neurons in a network will vary, forcing the model to control an efficient representation amongst all neurons. Third, linear separability. Sparse representations are either more likely, or more easily linearly separably, leading to better model performance. Lastly, distributed but sparse representations. Sparse

representations force the model to distribute information across the network as at any moment neurons can be turned off by their input.

Despite the success of ReLU activations, it suffers from another phenomenon referred to as the dying ReLU problem, which occurs when most the neurons output zeros, thus leading to more inactive neurons in subsequent layers. In addition, the derivative of the ReLU is defined to be zero when the input is less than zero. As a result, when most of the neurons become inactive, so do their gradients, leading to poor learning. To alleviate this problem, a family member of the rectified linear units referred to as leaky rectified linear unit, Leaky ReLU, has been established [34].

Leaky ReLU, is equivalent to ReLu, but instead of outputting zero when the input is less than zero, it leaks information by having an extremely small slope for negative values, as can be seen by multiplying the input by $\alpha$, in Equation 2.21.

$$f(net) = \begin{cases} net & net > 0 \\ \alpha * net & net < 0 \end{cases} \tag{2.21}$$

Another member of the rectified family created to solve this issue is the exponential linear unit, ELU [32]. When the input is negative, instead of having a small negative slope like Leaky ReLU, ELU has an exponential decay ranging from -1 to zero, given in Equation 2.22.

$$f(net) = \begin{cases} net & net > 0 \\ \alpha * (e^{net} - 1) & net < 0 \end{cases} \tag{2.22}$$

A close relative to ELU is the scaled exponential linear unit, SELU [15]. SELU, like ELU, utilizes an exponential when the input is negative and returns

the identity when the input is positive; however, SELU, scales ELU by a constant, $\lambda$, as can be seen in Equation 2.23. In the original paper, the $\lambda$ and $\alpha$ were empirically found to be 1.0507 and 1.6733.

$$f(net) = \begin{cases} \lambda * net & net > 0 \\ \lambda * \alpha * (e^{net} - 1) & net < 0 \end{cases} \tag{2.23}$$

Altogether, the plots for all activation functions from the rectified family discussed thus far are depicted in Figure 2.19:

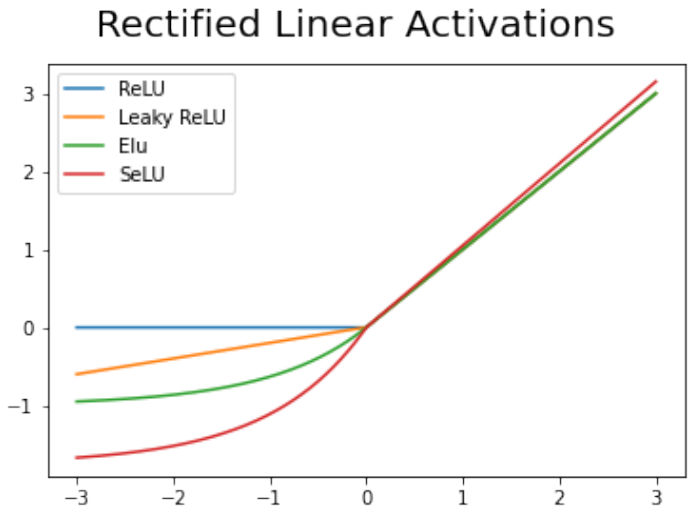



Figure 2.19: Rectified Family

Unlike the previous activation layers which are applied to hidden dense or convolution layers, the softmax activation layer is the activation applied to the last output layer. The softmax activation function normalizes the sum of exponential from its input, creating a probability distribution, as shown in equation 2.24.

$$f(net) = \frac{e^{net}}{\sum_i e^{net_i}} \tag{2.24}$$

Because the softmax layer creates a probability distribution, it can be applied directly to a cross-entropy loss function for evaluation. The benefit of utilizing softmax over simply standardizing the output for creating a probability distribution, is that the exponential of the softmax function cancels out the logarithm in cross entropy, allowing for the loss to be roughly linear with respect to its input. As a result, the loss will never saturate, leading to efficient learning [10].

### 2.6.6 Generalization

In Section 2.2.1 on overfitting and underfitting, the phenomenon known as overfitting was discussed. Overfitting occurs when the model performance on the training data greatly exceeds the performance of the model on the test or validation data. Generalization refers to when the model performance on the testing or validation data is near that of the training data or better. Overfitting can occur when the model complexity is too large, i.e. it has too many trainable parameters, and the model starts to memorize the input data; or, when the model weights start to become too large for a small set of neurons. There have been many approaches taken to increase generalization, but the methodologies that will be focused on here will be weight penalization, early stopping, neuron dropping, and data augmentation.

#### L1 and L2 Regularization

Weight decay is a simple way to penalize models for having large weights. Two prominent methodologies have been established, Lasso (L1) and Ridge (L2) [18].

Both L1 and L2 add a penalization term to the error function that increases when the magnitude of the weights of the model increases. The difference between L1 and L2 is that the L1 penalty term is built off the sum of the absolute values of the weights, as can be seen in Equation 2.25, while the L2 penalty term is built of the sum of the squared values of the weights, as can be seen in Equation 2.26, where $\mathcal{L}(w)$ refers to the loss function given the weights of the model, and $w_i$ refers to each individual weight.

$$\mathcal{L}(w) + \lambda \sum_i |w_i| \tag{2.25}$$

$$\mathcal{L}(w) + \lambda \sum_i w_i^2 \tag{2.26}$$

Both L1 and L2 have a coefficient, $\lambda$, that defines how much influence the penalty term has in regards to the error function. Too large of a value will lead to underfitting, as the weights will become too small to learn any representation, while too large of a value will lead to overfitting, as the weights will become too large. Because L2 penalizes the size of the coefficients, the overall final weights will be comparatively small. On the other hand, because L1 penalizes the magnitude of the coefficients, some of the weights will be driven towards zero, creating sparse weights.

### Early Stopping

A very similar approach to L2 weight decay is early stopping. Early stopping prevents the model from having too large of weights by stopping training when the model starts to overfit. Typically, the test or validation set is given to the model during training. After each epoch, the test or validation set is evaluated for its

generalization capabilities. Whenever the performance on the test or validation data starts to decline, indicating overfitting, as the model weights are becoming too large, the training stops in order to preserve the smaller weights.

### Dropout

Dropout [27] is a layer introduced to reduce overfitting, not by adding a penalization term to the loss function but instead by dropping neurons, along with their weights, during training. With each pass throughout the network, a randomly sampled set of neurons, along with their associated weights, are dropped, simulating sparsity. The only hyper parameter needed for a dropout layer is the dropout percentage, which refers to the percentage of nodes between two layers to be randomly dropped. By randomly dropping neurons and weights, the model is forced to not rely upon a set of weights with large values, but now must rely upon a randomly sampled set of weights. Dropout also increases model robustness as each training pass throughout the network samples a random set of weights. As a result, each training pass through the network relies upon a different randomly sampled subnetwork, which decreases inter-neuron dependencies that cause overfitting.

### Batch Normalization

Another layer utilized for increasing generalization is batch normalization. Unlike the previous techniques utilized for increasing generalization, batch normalization [13] was mainly designed to solve a phenomenon in activation functions known as internal covariate shift. Each convolution and dense layer will have its own unique input distribution. Whenever this distribution changes during training it is referred to as covariate shift. Internal covariate shift occurs when the distri-

bution of network activations changes due to the weight updates of the model. This internal covariate shift can slow down training as the entire model has to compensate for internal distribution shifts at different layers within the network. Batch normalization was proposed to solve this issue by standardizing the input before being sent to the activation layer. The standardization works by simply transforming the input into Z-scale for each dimension, where the mean and variance are calculated for that particular mini-batch sample, as shown in Equation 2.27, where $k$ is the $k^{th}$ dimensional input, $E$ is the mean and $V$ is the variance of the mini-batch sample.

$$\hat{net}^k = \frac{net^k - E[net^k]}{\sqrt{Var[net^k]}} \tag{2.27}$$

In addition to speeding up training, the authors found that applying batch normalization also increased generalization.

### Image Augmentation

Another methodology of increasing the generalization of a network is through data augmentation, which refers to the process of augmenting the training data by diversification. As a result, data augmentation creates artificially realistic data that the model can learn underlying representations from. This process is more commonly applied to convolutional neural networks as image augmentation is typically more intuitive than when the data is not an image.

From the success of L1 regularization, dropout, and ReLU activation as a means of creating robustness and regularization through sparsity, Cutout is a simple regularization technique that drops a patch of pixels from the input image prior to being fed into the model. Unlike dropout, which drops neurons ran-

domly during training, Cutout drops the pixels before even being processed by the model. Cutout was shown to improve regularization over baseline models [4].

Overfitting can occur when the model starts to memorize pixel representations and can become very sensitive to noise. Another form of image augmentation created to solve this issue is mixup [2]. Mixup changes the images by creating convex combinations between images. Because the input image is no longer resembling only one class, the output label is adjusted in order match the linear combination between the images.

The last image augmentation technique to be discussed combines both Cutout and Mixup, referred to as cutmix [35]. The downside of Cutout is that vital information of the input image is left out as a randomly selected patch is dropped, while the problem with Mixup is that the samples generated are unrealistic and can cause ambiguity in the transformed training labels. Cutmix solves both problems by selecting a random patch of an image and replaces that patch with the same patch from another image. CutMix was shown to outperform both Cutout and Mixup on numerous data sets and model architectures.

Examples of these augmentation techniques can be found in Figure 2.20. Part *a* and part *b* of Figure 2.20 showcase the original two images from an example dataset. Part *c* performs Cutout on the original cat image by replacing a randomly patched box with zeros. Part *d* performs a linear combination between the cat and dog for Mixup. Lastly, part *e* showcases Cutmix, where a randomly patched box from the cat image is replaced with the associated box coordinates from the book image.

(a) Original Picture for Cat

(b) Original Picture for Book

(c) Cutout Example

(d) Mixup Example

(e) CutMix Example

Figure 2.20: Image Augmentation Techniques

### 2.6.7 Optimizers

Training and optimizing deep CNN pose a challenge for both efficiency and performance. The ability to converge fast is desirable in order to reduce computational costs, but also the desire to converge to minima points that yield better performance is even more desirable. The optimization landscapes of deep neural networks are vast and complex terrains with steep cliffs, saddle points, plateus, and valleys [10]. Being able to maneuver across this landscape is vital in order to achieve better performance. Over the history of deep learning, many different optimizers have been created. Within the context of this work, variants of SGD will be discussed along with AdaGrad, RMSProp, and Adam. Lastly, weight decay for both SGD and Adam will be introduced as a methodology for generalization.

#### SGD and Momentum

For reiteration, SGD across a single batch is given by

$$w_{i+1} = w_i - \alpha \frac{1}{n} \nabla \sum_{j}^{n} F(w)|_{w=w_i} \tag{2.28}$$

Because SGD updates a single batch across the gradient average instead of the gradient of the full dataset, each update step can become noisy in the sense that it takes a non-perfect step. In addition, SGD can become stuck in local ravines of the optimization landscape. In order to escape from such local ravines and take better steps, momentum can be applied. Momentum can be explained like a ball rolling down a hill. As the ball keeps rolling down the hill, it picks up momentum and speed which helps it reach the bottom quicker. Just like this simple analogy, momentum incorporated with SGD can speed up convergence and escape local minima; however, it can also become sporadic and bounce out of an optimal

critical point if it becomes too large. Momentum is implemented by updating the weights by the weighted subtraction of the average moving gradient, given in equation 2.29.

$$\begin{aligned} v_{i+1} &= \gamma v_i + (1-\gamma)\alpha \frac{1}{n} \nabla \sum_{j}^{n} F(w)|_{w=w_i} \\ w_{i+1} &= w_i - v_{i+1} \end{aligned} \tag{2.29}$$

A velocity term is kept and updated at every time step, as in PSO. The $\gamma$ parameter represents the momentum hyperparameter. Controlling this hyperparameter influences how much the previous velocity will impact the weight update. Algorithms which utilize momentum incorporate what is known as moving averages, the average of past gradients that move due to iteration updates. From Equation 2.29, each velocity update is a linear combination of the current gradients plus the previous gradients, $v_i$. With this incorporation, momentum acts a source of accumulation in order to utilize the history of gradients in calculating the next step.

A variation of momentum is known as Nesterov Momentum [21],

$$v_{i+1} = \gamma v_i + (1-\gamma)\alpha \frac{1}{n} \nabla \sum_{j}^{n} F(w)|_{w=w_i+\alpha * v_i} \tag{2.30}$$

Nesterov and standard momentum are equal except for the calculation of the gradient. Nesterov calculates the gradient with respect to the projected position, $w_i + \alpha v_i$. By calculating the gradient with respect to the projected position, the gradient influences the direction of where to step based upon that projection.

### AdaGrad

One issue with all variants of SGD is that the learning rate is static for all parameters. Although this rate can be increased or decreased, it is increased or decreased for all parameters. This can become problematic because learning rates with very small values will take much longer to convergence while learning rates with extremely large of can lead to oscillation and divergence. As a result, having a static learning rate for all parameters can limit the learning of some parameters while causing others to oscillate. In this way, it is desirable to have individual learning rates for each parameter in order for better optimization; however, manually setting millions of learning rates is infeasible. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization [5] proposed an algorithm called AdaGrad to solve this issue by adaptively scaling the learning rate for each parameter. A learning rate is still given to the algorithm to represent the maximum ceiling, but each parameter has the ability to scale up to the learning rate. The algorithm has the ability of changing the learning rate based on the sum of the squared gradients over the course of training. The algorithm can be seen in Equation 2.31, where the gradient of the $k^{th}$ parameter at time step $i$.

$$\begin{aligned} g_{i,k} &= \frac{1}{n}\nabla F(w)|_{w=w_i+\alpha*v_i} \\ w_{i+1} &= w_i - \frac{\alpha}{\sqrt{\epsilon + \sum_j^i g_{(i,k),j}^2}} g_{i,k} \end{aligned} \tag{2.31}$$

The learning rate is scaled by the denominator, the square root of $\epsilon$, the stability term, plus the sum of the squared gradients over the course of all iterations of training. As a result, parameters with large gradients over the course of training will scale the learning rate to decrease, while smaller gradients will yield an

increase in scaling of the learning rate. Because the denominator is dependent upon only the gradients of that particular weight, the each individual learning rate becomes scaled to adapt to training.

### RMSProp

Another issue with all variants of SGD is that the gradient average can have issues when the magnitudes of one gradient of the batch is extremely large, which can negatively affect optimization. Root Mean Squared Propagation, RMSprop, is an unpublished algorithm that seeks to fix this issue by normalizing the squared average of the gradients. Simultaneously, it also tries to solve the problem with AdaGrad where the adaptive learning rates for individual can pre-converge to zero before overall model convergence due to that fact that the gradients can be very large early on during training, resulting in the accumulation of gradients over time to be larger than what is reflected from recent gradients. In this way, AdaGrad has the ability only to adaptively decrease individual learning rates, but not increase. RMSProp solves both problems described above by taking a weighted moving average of the squared gradients from previous iterations and its current, then scaling the learning rate by that said value. The RMSProp algorithm can be given in equation 2.32, where $s_{i,k}$ is the moving average of gradients for the $k^{th}$ weight.

$$\begin{aligned} g_{i,k} &= \frac{1}{n}\nabla F(w)|_{w=w_i+\alpha * v_i} \\ s_{i,k} &= \gamma s_{i-1,k} + (1-\gamma) g_{i,k}^2 \\ w_{i+1} &= w_i - \frac{\alpha}{\sqrt{\epsilon + s_{i,k}}} g_{i,k} \end{aligned} \tag{2.32}$$

### Adam

The Adam optimizer, introduced in Adam: A Method for Stochastic Optimization [14], sought to combine RMSProp with momentum in order to accelerate convergence while also having the ability to adaptively increase and decrease the learning rate per parameter. The moving averages of the gradients and the moving averages for momentum are calculated by equation 2.33, then the current time steps momentum and moving average of gradients after calculation are normalized by two hyper parameters $\beta_1$ and $\beta_2$ raised to the $i^{th}$ power, as shown in equation 2.34. Finally the weights are updated in equation 2.35, with the noticeable difference in that $\epsilon$ is included outside the square root instead of inside, as in RMSProp and AdaGrad. However, Adam introduces now two very important hyper parameters to tune other than the max ceiling learning rate, $\beta_1$ controlling the momentum and $\beta_2$ controlling the moving average of gradients. In the original paper, the authors suggest $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

$$\begin{aligned} g_{i,k} &= \frac{1}{n}\nabla F(w)|_{w=w_i+\alpha*v_i} \\ v_{i,k} &= \beta_1 v_{i-1,k} + (1-\beta_1) g_{i,k} \\ s_{i,k} &= \beta_2 s_{i-1,k} + (1-\beta_2) g_{i,k}^2 \end{aligned} \tag{2.33}$$

$$\begin{aligned} \hat{v}_{i,k} &= \frac{v_{i,k}}{1-\beta_1^i} \\ \hat{s}_{i,k} &= \frac{s_{i,k}}{1-\beta_2^i} \end{aligned} \tag{2.34}$$

$$w_{i+1} = w_i - \hat{v}_{i,k}\frac{\alpha}{\epsilon + \sqrt{s_{i,k}}} g_{i,k} \tag{2.35}$$

### Decoupled Weight Decay

As discussed in section 2.6.6 over generalization techniques, L1 and L2 regularization were both explored as methodologies to regularize neural networks. Both regularization techniques added an additional cost metric to the calculation of the cost function. In the current discussion on optimizers, this would be applied during the calculation of the gradient. However, a problem arose with adaptive optimizers, such as Adam, where the L1 and L2 regularization would become adapted for each individual parameter, negating the intent. Despite applying weight regularization directly to the loss function, another methodology of weight decay has been proposed for adaptive optimizers. In Decoupled Weight Decay Regularization [20], the authors propose a weight decay mechanism that is decoupled from the loss function and is directly applied to the weight update step. The Adam optimizer incorporated with decoupled weight decay is commonly referred to as AdamW. AdamW changes the update step for each parameter, which can be seen in equation 2.36. Instead of multiplying the gradient directly, AdamW scales the previous weight position by the weight decay coefficient, $\lambda$, before adding it to the rest of the expression. As a result, decoupled weight decay regularizes all weights equally by scale $\lambda$ in order to prevent weights from becoming too large, leading to overfitting.

$$w_{i+1} = w_i - \hat{v}_{i,k}\frac{\alpha}{\epsilon + \sqrt{s_{i,k}}} + \lambda w_i \tag{2.36}$$

### 2.6.8 Advanced Convolutional Neural Networks

#### AlexNet

In order to compare new and upcoming CNN architectures, universal benchmark image recognition datasets were established for accurate comparison. The most staple benchmark utilized to compare all major models is ImageNet LSVRC-2010, which comprises of 1.2 million high-resolution images across 1000 unique classes. One of the first successful ConvNets on this benchmark dataset was an architecture called AlexNet [17], created in 2012. The AlexNet architecture contained only eight simple convolution layers, three max pooling layers, three dense layers, and one dropout layer, which can be seen in Figure 2.21.

The first convolution layer contained a kernel with size 11x11, followed by a max pooling layer, then a kernel with size 5x5, max pooling layer, and then the remaining convolution layers had a kernel size of 3x3 followed by a max pooling layer. The original idea behind the decreasing kernel sizes was to capture spatial representation proportional to the input size. After each convolution and dense layer, a ReLU activation function was applied.

Although, this relatively small architecture contained only a few convolution layers, the final model totaled to 62.3 million trainable parameters and achieved 62.5% test accuracy on the ImageNet benchmark dataset. Since the emergence of AlexNet, a plethora of different CNN architectures have been explored, from VGGNet and ResNets to Inception and Xception.

#### VGGNet

Shortly following AlexNet, VGGNet networks emerged [25], extended the number of layers in AlexNet to a much larger scale. VGG networks follow the same pro-

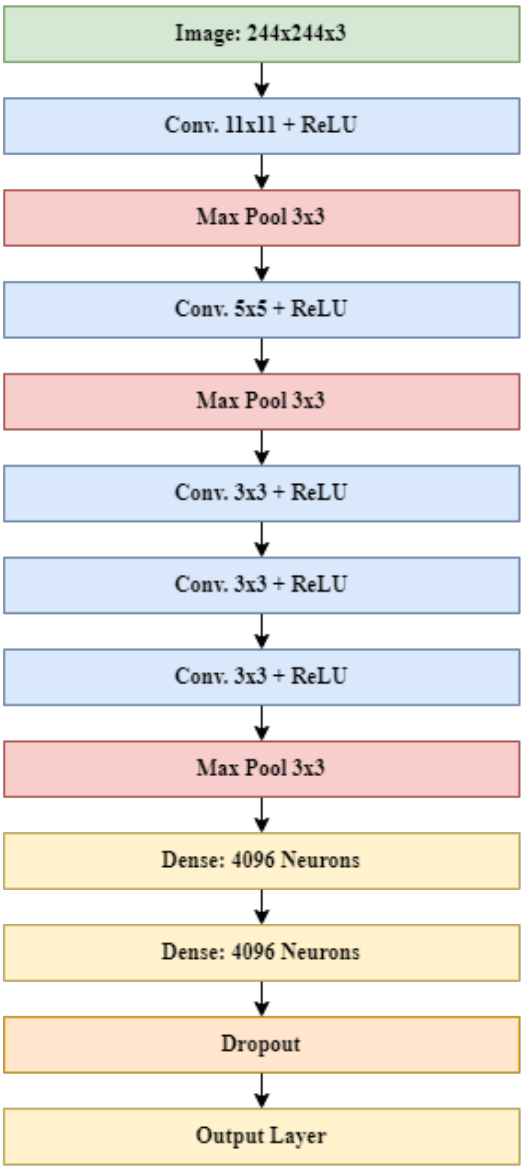



Figure 2.21: AlexNet Architecture

cedure as AlexNet, a feed forward network composed of blocks of convolutional kernels where each block is followed by a max pooling layer for dimensionality reduction, wrapping up with three dense layers. Another unique difference between AlexNet and VGG is that the authors of Very Deep Convolutional Networks showcased that large kernel sizes of 11x11, 7x7, and 5x5 can be reduced by stacking

multiple 3x3 convolutional layers. By reducing the kernel size and stacking more kernels, the number of trainable parameters is reduced dramatically while also not losing any model performance.

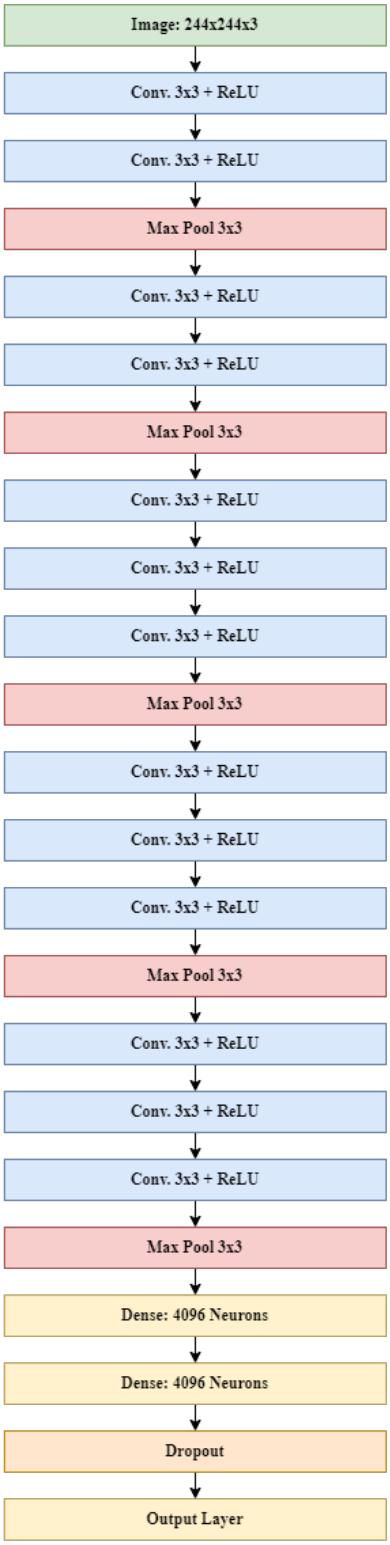
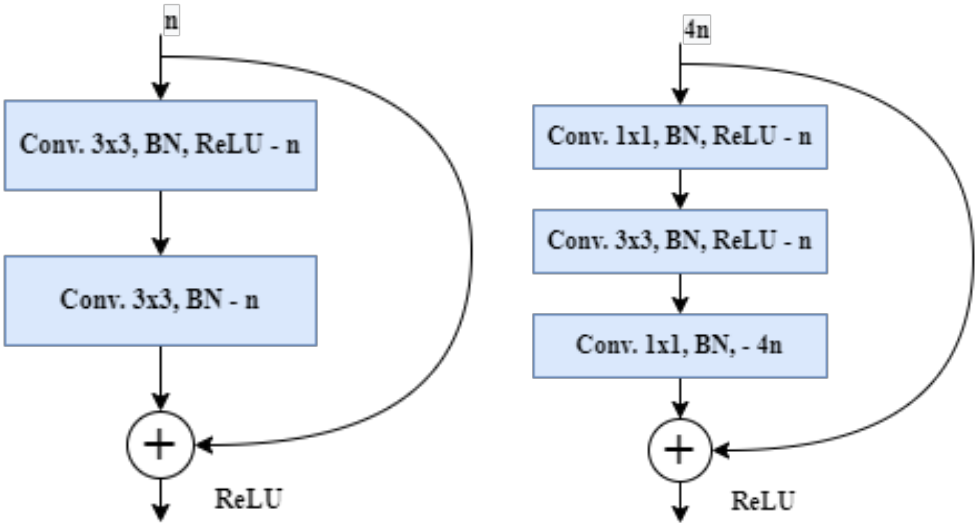



Figure 2.22: VGG16 Architecture

Two prominent models were created, VGG16 and VGG19, where the numeric represented the number of convolutional layers plus dense layers. The VGG16 architecture can be seen in Figure 2.22. Together, VGG16 contained 138.4 million parameters and VGG19 contained 143.7 million parameters while both yielding a new best test accuracy for the time of 71.3% on ImageNet.

### ResNet

Shortly following after VGG19, deep residual connections, commonly referred to as ResNets were created [12]. The authors of Deep Residual Learning for Image Recognition incorporated many new ideas into their models: skip connections, bottleneck modules, global average pooling, and batch normalization. The authors observed a phenomenon where the test and training error was worse for networks containing a very large number of convolution layers, up to 200. They believe the reason behind this occurrence is due to the fact that extremely deep convolutional networks are too difficult to optimize as information becomes lost throughout the network. They addressed the issue by incorporating deep residual learning via skip connections that passed information from one layer to another by skipping subsequent layers, which can be seen in Figure 2.23$a$, where $n$ refers to the number of filters, and the plus symbol refers to layer wise element addition.




(a) Residual Module (b) Residual Module with Bottleneck

Figure 2.23: ResNet Modules

The architecture was broken down into modules that were stacked repeatedly. Each module contained two convolutional layers with kernel sizes of 3x3, followed by a ReLU activation after each layer. In addition, following after each convolution, but right before the activation, a batch normalization layer was added in order to reduce the likelihood of internal covariate shift. Unlike AlexNet and VGG, ResNets achieved dimensionality reduction by utilizing a convolution layer with a stride of two instead of the cheaper max pooling layer.

In addition to skip connections, Deep Residual Learning for Image Recognition also proposed a bottleneck module. Even after reducing kernel sizes, the computational costs for two stacked 3x3 kernels is dependent on the number of projecting channels from the previous layer. The motivation behind the bottleneck module is to reduce this computation. The bottleneck module is a building block that contained three convolution layers, the first and last with a kernel size of 1x1 and the middle with a kernel size of 3x3, as can be seen in Figure 2.23*b*, where $4n$ refers to four times the amount of filters for that module.

The smaller kernel size of 1x1 offers a cheap computation that will reduce and restore the information within the module. The first 1x1 kernel will compress the information fed into the 3x3 kernel by decompressing the number of channels, while the last 1x1 kernel will restore the information by projecting it back into the original feature space by increasing the number of channels. The purpose of this compression and decompression is to not only reduce computational costs, but also extract the most important features. Although this compression and decompression might seem to lose information, the combination with skip connections incorporates previous uncompressed information with the current helps prevent any major information loss.

Lastly, unlike AlexNet and VGG, ResNets utilized a single global average

pooling layer instead of dense layers before the final dense output. By utilizing global average pooling, the number of parameters decreases greatly while also not degrading performance. Like average pooling, global average pooling performs dimensionality reduction by averaging pixels, except across the channels instead of the height and width. For example, an input of 16x16x512 is reduced down to a 1x1x512 as each feature map is reduced via averaging.

By varying the number of repeated modules per model, ResNet model sizes can grow dependent upon the computational resources available. On the ImageNet benchmark dataset, ResNet152, where the numeric refers to the number of total layers, achieved a result of 76.6% accuracy while only having 60.4 million parameters. For a full example of a ResNet architecture, Figure 2.24 show the ResNet50 architecture where Block64x3 refers to the bottleneck module described in Figure 2.23*b* where $n = 64$ and the module is repeated three times.

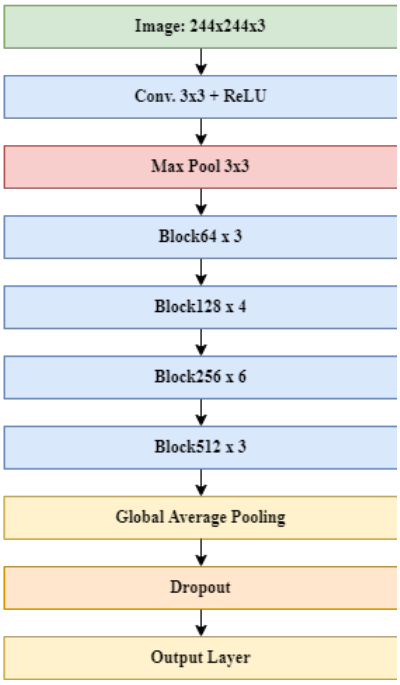



Figure 2.24: ResNet50 Architecture

### Inception V1

Around the time of ResNets, an architecture referred to as Inception [30] took a very different approach to constructing convolutional neural networks. Unlike VGG and ResNets, Inception challenged the idea of creating very large deep neural networks. Instead, Inception brought the idea of shallower models with more layers clustered together per module. Creating larger models by stacking more layers or increasing the number of channels leads to two prominent issues, over fitting and increased computational cost. Inception proposed a way to solve both problems by creating sparse architectures. Their research showed that combining relatively sparse information into dense information can lead to increased performance. With this theoretical research as a basis, Inception proposed the revolutionary idea of clustering layers together and combining their outputs as input into the next module. This can be seen in Figure 2.25. The final architecture is created by stacking multiple inception modules up to the computational resources available.

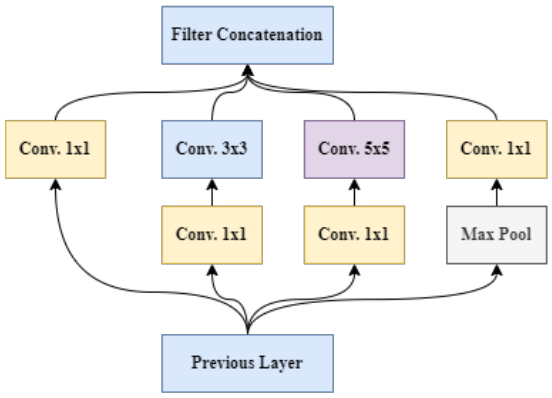



Figure 2.25: InceptionV1 Module

In Figure 2.25, the current module has four output connections fed into the next module. The first is from a 1x1 convolution, while the second is a 3x3 convolution preprocessed by a 1x1 convolution, the third is a 5x5 convolution preprocessed by a 1x1 convolution, and the last connection is from a max pooling layer with a stride of one that is followed by a 1x1 convolution. The intuition behind the different kernel sizes in the module was to capture different spatial representations of the image. The output from the four layers was combined using filter concatenation, which combined the output by stacking them via the filter/channel dimension. Although this increased the number of channels for the next module, the cheap 1x1 convolutions preceding three of the four connections for the next module allowed for compression back to a smaller dimension.

This module was repeated a number of times where it was then followed by max pooling for dimensionality reduction. By repeating this process, the final model was created. Like ResNets, Inception utilized ReLU after each convolution layer and global average pooling right before the output layer. This architecture was known as InceptionV1, for version one.

### Inception V2-V3

Versions two and three of Inception were released in a single paper called Rethinking the Inception Architecture for Computer Vision [31]. InceptionV2 reduced the computational costs of InceptionV1 by reducing the 5x5 kernel into two stacked 3x3 kernels. In addition to this reduction, InceptionV2 utilized factorization to achieve even smaller network sizes without diminishing performance. Stacked convolutional kernels can be factorized to asymmetric convolutions, $n$x1 convolutions. For example, a 3x3 convolution would be factorized to a 1x3 and 3x1. By factorizing the kernels, the computational requirements were reduced as applying

a 1x3 kernel followed by a 3x1 kernel is more computationally efficient than a 3x3. See Figure 2.26 for the construction of the new module.

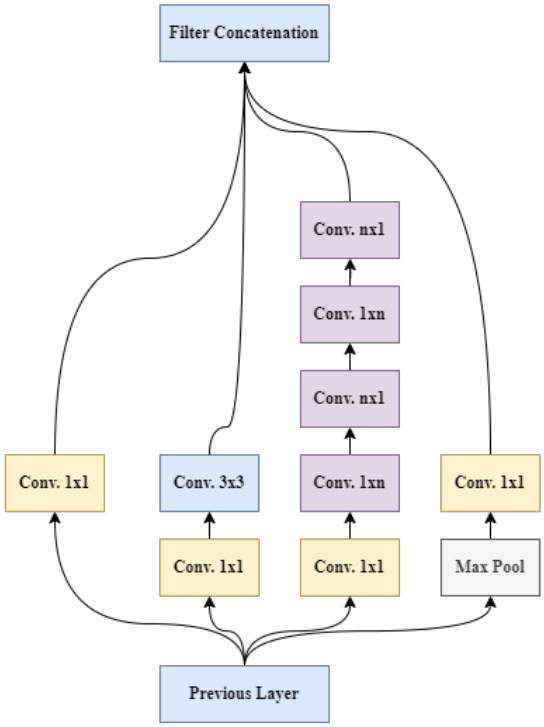



Figure 2.26: InceptionV3 Module

Lastly, like ResNets, InceptionV2 switched from applying max pooling for dimensionality reduction after modules as in InceptionV1, to applying a stride factor of 2 for each of the convolution layers in a module.

InceptionV3 expanded on InceptionV2 with the addition of regularizes, such as batch normalization layers between kernels, dropout layers, and label smoothing. Label smoothing is the methodology of increasing regularization by making the model less confident concerning its predictions, reducing over fitting. The authors showed that two problems can arise with using simple cross entropy; the first being that if a model can learn to assign extremely large probabilities to ground truth labels during training, which can decrease generalization; secondly, because the gradient of cross entropy with respect to the input distribution is bounded between -1 and 1, the ability for the model to adapt is reduced if it becomes too confident concerning the learning samples. As a result, label smoothing reduces the confidence of the largest quantile from the predicted distribution while increasing the probabilities of the rest of the quantiles to a small degree. The exact alteration of a predicted distribution $P$ for an $i^{th}$ observation is given in Equation 2.37, where $\alpha$ is the source of degrading confidence and $K$ is the total number of classes. As an example, assuming a predicted distribution for a observation to be $P_i = (1, 0, 0)$ with 3 classes and $\alpha = 0.10$, the new predicted distribution to fed into cross entropy will be $\hat{P}_i = (0.933, 0.033, 0.033)$, which decreases the confidence of the current largest prediction while also giving a small boost to other classes.

$$\hat{P}_i = (1 - \alpha)P_i + \frac{\alpha}{K} \tag{2.37}$$

Lastly, InceptionV3 changed the module from InceptionV2 by using factorized

7x7 kernels instead of 3x3. Even though a single 7x7 kernel is more computationally complex than a 3x3, factorizing the kernel into 1x7 and 7x1 kernels reduced the complexity by 33%. InceptionV3 achieved 77.9% accuracy with only 23.9 million parameters on the ImageNet benchmark dataset.

### Inception V4, Inception-ResNet, and Xception

InceptionV4 and Inception-ResNet were released again in a single paper called Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning [29]. Although the exact additions for InceptionV4 will not be discussed, Inception-ResNet combined Inception style modules with residual connections. The architecture was changed by adding a 1x1 convolution after the filter concatenation to compress the channels to its original size to then be added to the previous modules output. See Figure 2.27 for the Inception-ResNet module. Inception-ResNet achieved 80.3% accuracy with only 55.9 million parameters on the ImageNet benchmark dataset.

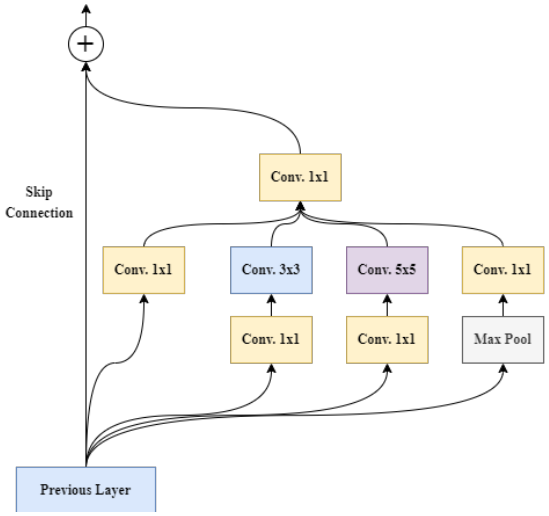



Figure 2.27: InceptionV3 Module

The last member of the Inception family is Xception, introduced the paper Deep Learning with Depthwise Separable Convolutions [1]. The architecture of Xception is fairly simple, the exact same as InceptionV2 except each convolution layer was swapped for a depth-wise separable convolution layer. The original hypothesis of Inception was to introduce sparsity spatially by combining the output from multiple convolutional kernels. Xception expanded upon this hypothesis by introducing sparsity not only spatially but also temporally via the channels. This effect can be captured by using depth wise separable convolutions, which separates the convolutional kernel into its depth wise and spatial kernels to be applied separately to the input. Xception achieved 79% accuracy with only 22.9 million parameters on the ImageNet benchmark dataset.

## 2.7 Neural Architecture Search

Current state of the art CNN architectures have mainly been hand designed by practitioners and researchers alike. Although hand designing architectures allows for the implementation of current domain knowledge, it closes the door to other possible types of architectures which could achieve better performance. The domain of exploring neural network architectures using some type of automated process can be referred to as Neural Architecture Search (NAS). NAS systems are composed of three primary functions: search space, search algorithm, and evaluation strategy [19].

### 2.7.1 Search Space

The search space defines the domain from which possible architectures can be constructed. Too large of a search space can lead to great exploration but poor

convergence qualities, while too small of a search space can lead to simple local greedy searches with poor exploration. There have been three primary search spaces for NAS systems: global, micro, and hierarchical. Global search spaces include search domains where no user bias is induced and every hyperparameter and possible combination of layers and connections is decided by the search algorithm. In global search spaces, the learning rate, optimizer, number of channels, kernel size, stride, regularization, number of layers, connections between layers, and other hyperparameters are all decided by the algorithm. Micro searches utilize user domain knowledge in order to limit the search space to possibly efficient architectures rather than completely random. The most common type of micro search utilizes cell-based modules that are designed by the search algorithm, which are combined and repeated to create the full model. Lastly, hierarchical search domains incorporate non-repeating cells that are built on top of each other as a non-cyclic directed graph.

### 2.7.2 Search Algorithm

The search algorithm defines how the NAS system will be able to maneuver about the search space. The NAS search space can be translated in to an optimization problem where the input is the created model and the output is the performance on some benchmark dataset. There are four common types of search algorithms in NAS systems: evolutionary algorithms, Bayesian optimization, reinforcement learning, and one-shot methods. As explained in section 2.4, evolutionary algorithms are population-based search algorithms which incorporate mutation and crossover in order to produce better individuals. Bayesian optimization creates a predictive model, samples from its prior beliefs concerning possible models, and

then updates the prior based upon the sampled results. Reinforcement learning is a sub-field of machine learning which is very similar to supervised learning. Unlike supervised learning, the label output is regarded as a reward, from which the reinforcement learning algorithm learns to maximize. For NAS search spaces, typical reinforcement models utilize a recurrent neural network to generate architectures and optimizes the weights using a standard reinforcement learning algorithm. Lastly, one-shot methods create a large hyper-graph of all possible combinations of the search space, training this extremely large model for a couple of iterations. After convergence, different connections and layers, along with their associated finished weights, are randomly sampled and evaluated.

### 2.7.3 Evaluation Strategy

The evaluation strategy defines the function that will grade each inputted model. The issue that arises with NAS systems for evaluating models is the fact that each inputted model needs to be trained on a given benchmark dataset for comparison. In this case, two datasets are primarily utilized, one for training the model and a second for validation. The best validation accuracy, or loss, is then utilized as the final score for the model. However, due to the nature of stochastic non-deterministic training, each model might yield a different final validation score from trial to trial, adding to the complexity of the optimization problem. In addition, training each model until convergence will yield the most accurate results, but can become extremely time consuming when the models and datasets are very large. Besides training until convergence, a few possible alternatives have been proposed in order to reduce training time. The simplest way to limit time complexity would be to limit the number of training epochs, risking the chance

of models not converging well, or train on a smaller subsample dataset. Another option is to internally share weights between models by using another models weights as the currents starting initial weights. Lastly, predictive models can be used in order to estimate the performance of models based upon certain criteria.

The benchmark image dataset discussed thus far has been ImageNet, but training a model upon this extremely large dataset can take days. As a result, it is common for NAS systems to evaluate models on much smaller and somewhat easier datasets, such as the CIFAR10 dataset. The CIFAR10 dataset is composed of 60,000 32x32 RGB images spanning over 10 unique classes. 50,000 of the images are split for training and 10,000 are split for testing. It is common for NAS systems to evaluate models on this smaller dataset before taking the best for testing on ImageNet.

# Chapter 3

# Related Work

The purpose of Chapter 2, Background Information, was to introduce the necessary information for the reader to understand the concepts and material being manipulated throughout this work. Chapter 3, Related Work, introduces three very vital and important papers that layout the map from which this work received inspiration from. First, the NASNet search space will be introduced, a revolutionary domain space that incorporated material from ResNets, Inception, and Xception, along with the ability to scale models after the searching process. Secondly, AmoebaNet will be introduced in order to showcase the current research utilizing genetic algorithms to maneuver about the NASNet search space. Lastly, Super Convergence will be described in order to demonstrate how the convergence of neural networks can be sped up.

## 3.1 NASNet

The design of the search space for NAS systems is vital in order to create powerful but yet unique architectures. In order to narrow down the search space

to architectures practitioners know will perform well, domain knowledge can be utilized in the creation of the search space. In, Learning Transferable Architectures for Scalable Image Recognition [38], the authors propose a search space that would revolutionize the NAS domain, the NASNet search space, a micro cell-based search space. The most powerful component of the NASNet search space is its scalability. Scaling CNN architectures poses a difficult task, as some practitioners allow the scaling of the model to be determined during the architecture search; however, larger models during the searching process will naturally take more computational resources to evaluate and possibly over fit the dataset. Instead, the NASNet search space defines two primary cells â normal and reduction. The search algorithm only optimizes these two independent cells, which can then be stacked upon each other for scalability after the searching process. The stacking of the cells is designed such that the normal cell is repeated $n$ number of times, before being followed by a reduction cell; this process of stacking the normal cell, followed by a reduction cell, is repeated until the input dimension down samples to the desired size. The exact construction can be seen in Figure 3.1, where a normal block of cells is followed by a reduction, then a normal block, reduction, normal block, and then output stem.

The number of filters per convolution layer in each cell is determined by the stage of the model in which the cell lies. After each reduction cell, the number of filters per layer doubles to match the spatial dimensionality reduction. For example, the first block of normal cells after input could have 32 filters, then the next block of normal cells after the first reduction cell will have 64, and then the final block will have 128. As a result, scalability can not only go depth wise, but also width wise by increasing the number of initial filters.

Both the normal and reduction cells receive two input connections and outputs

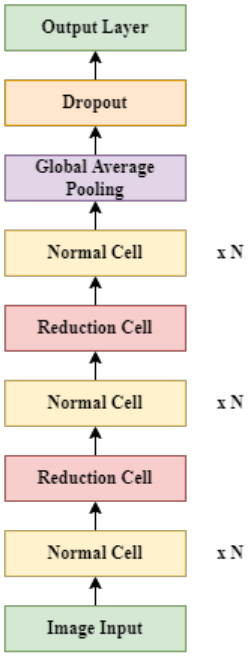



Figure 3.1: NasNet Model Construction

one single connection. The first input connection is the output from the previous cell while the second input connection is the skip connection input from the second previous cell, which can be seen by Figure 3.2.

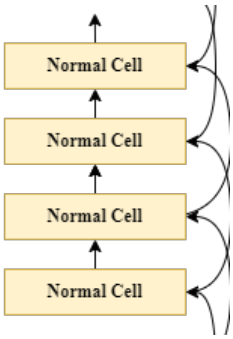



Figure 3.2: Normal Cell Module with Residual Connections

This skip connection input was incorporated due to the success of ResNet architectures, as residual connections allow for older information to pass throughout the network. The output connection is the filter concatenation from a set of hid-

den states. The hidden states are Inception like building blocks containing multiple hidden states, or nodes, that receive two connections and applies one unique operation to each input. The number of hidden states is defined by the architect, but the search algorithm has the ability to define the topology of the connections and which operations to apply to each incoming connection. The only difference between the normal and reduction cell, besides being optimized independently, is that the reduction cell utilizes a stride of two at each first input connection in order to achieve dimensionality reduction. An example construction for a cell containing 5 nodes is given in Figure 3.3.

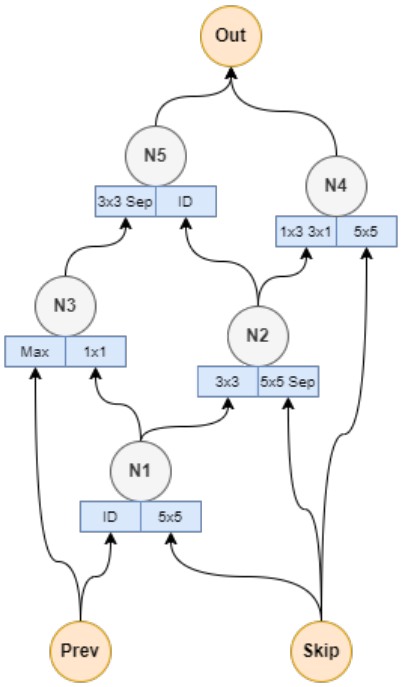



Figure 3.3: Normal Cell Example

The list of possible operations to be applied at each hidden state are hand-picked from current state of the art research. In the NASNet search space, 13 possible operations were given, listed in Table 3.1.

The NASNet search space combined scalability with the domain knowledge

Table 3.1: **NASNet Search Space Operations**

| Operations | |
|---|---|
| Identity | 1x3 3x1 Conv. |
| 1x7 7x1 Conv. | 3x3 Dilated Conv. |
| 3x3 Avg. Pool | 3x3 Max Pool |
| 5x5 Max Pool | 7x7 Max Pool |
| 1x1 Conv. | 3x3 Conv. |
| 3x3 Sep. Conv. | 5x5 Sep. Conv. |
| 7x7 Sep. Conv. | |

of residual connections and inception like building blocks in order to produce a search space with the ability to create powerful but yet unique architectures. For the final details of the search space, all convolution layers followed the order of an ReLU activation, convolution operation, and then batch normalization. However, when depth wise separable convolution was employed as an operation, two were applied to the connection instead of one. In addition, when the operation was depth wise separable convolution, no batch normalization was applied between the two separable convolutions. Lastly, in order to satisfy dimensionality consistency, 1x1 convolution layers were inserted as necessary.

After designing the search space, the search algorithm selected was a reinforcement learning recurrent neural network controller to generate the model architectures, updating the weights of the control using Proximal Policy Optimization [24]. Finally, the evaluation strategy created to score each model was to train each model on the CIFAR10 dataset, utilizing a randomized subsample of 5000 images from the 50,000 training images as validation. Each model was scaled to $n = 4$ and trained for 20 epochs with a batch size of 128 on the remaining 45,000 training images, equating to approximately 7,000 mini-batch weight updates. The searching process utilized 500 GPUs over four days of optimization,

resulting in 2,000 total GPU hours, and evaluated a total of 20,000 models. The best model, termed NASNet-A, was then scaled to $n = 6$ with 768 filters in the penultimate layer and $n = 7$ with 2304 filters in the penultimate layer, on the CIFAR10 dataset, their results can be seen in Table 3.2. The penultimate layer is the final global averaging layer before being fed into the output layer.

Table 3.2: **NASNet-A Test Accuracy on CIFAR10 Dataset**

| Architecture | Parameters (Millions) | Test Accuracy |
|---|---|---|
| NASNet-A (6@768) | 3.30 | 97.35% |
| NASNet-A(7@2304) | 27.60 | 97.60% |

## 3.2 AmoebaNet

With the success of Learning Transferable Architectures for Scalable Image Recognition, other researches decided to tackle the newly designed NASNet search space, except from the side of evolutionary computation. In Regularized Evolution for Image Classifier Architecture Search [22], the authors proposed an evolutionary algorithm with two core components which achieved very similar results with the original reinforcement learning controller. The authors utilized the same NASNet search space, except limited the number of operations to only eight operations, getting rid of almost all standard convolution operations in favor of separable convolution, listed in Table 3.3.

The genetic algorithm proposed by the authors was a population-based mutation only algorithm with aging. The algorithm randomly initialized a population, sampled from the population with replacement, and utilized tournament style selection to choose the best model from the sample to be mutated. After mutation, the new individual was then added back to the population and the oldest

Table 3.3: **AmoebaNet Search Space Operations**

| Operations | |
|---|---|
| Identity | 3x3 Sep. Conv. |
| 5x5 Sep. Conv. | 7x7 Sep Conv. |
| 3x3 Avg. Pool | 3x3 Max Pool |
| 3x3 Dilated Sep. Conv. | 1x7 7x1 Conv. |

from the population was removed. Because training each CNN architecture is non-deterministic, the authors believe that holding onto the best individual can diminish performance if the model was incapable of repeating its performance. As a result, the only way for a model to survive was if it were passed down to its child through mutation. By incorporating aging, the genetic algorithm was less prone to early convergence. The genetic algorithm only mutated an individual with one of two possibilities, randomly changing the connections or operations of a node. The authors did not state as to why crossover was not utilized. The authors ran their evolutionary algorithm with five different configurations, changing the population size and the number of samples to drawn for tournament selection, for comparison. In their results, the authors report that a population size of 100 individuals with 25 randomly sampled for tournament selection performed the best. Each trial run for evolution evaluated a total of 20,000 models after running 450 GPUs for 7 days. Each model was scaled to $n = 3$, training for 25 epochs on 45,000 of the training set, and used the rest of the 5,000 training images for validation. The best model after evolution was dubbed AmoebaNet-A, which was scaled to n=6 with 32 and 36 starting filters. Their results can be seen in Table 3.4.

Table 3.4: **NASNet-A Test Accuracy on CIFAR10 Dataset**

| Architecture | Parameters (Millions) | Test Accuracy |
|---|---|---|
| AmoebaNet-A (N=6, F=32) | 2.60 | 96.60% |
| AmoebaNet-A (N=6, F=36) | 3.20 | 96.66% |

## 3.3 Super Convergence

Learning rate schedules are common practice when it comes to training deep neural networks. Exponential decay is a type of learning rate scheduler that exponentially reduces the learning rate of an optimizer so that the step sizes become smaller for better convergence. One type of learning rate schedule has been shown to speed up the training time of large CNN architectures. In Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates [26], the authors propose a cyclic learning rate schedule that cycles once from a minimum to a maximum learning rate. Cyclic learning rates were based on the combination between curriculum learning and simulated annealing. By increasing the learning rate during training, the step size towards the gradient increases in magnitude, allowing for the algorithm to converge faster. In addition, by allowing the optimizer to have a larger learning rate, it has the ability to escape from local critical and saddle points. For an example of a single cycle learning rate schedule, Figure 3.4 depicts a cyclic learning rate schedule from 0.001 up to 0.10 and back down again over the course of 90,000 mini-batch weight updates.

However, setting the maximum learning rate is another hyper parameter to tune, as it was shown to be dependent upon the model, dataset, and regularization approaches taken. Too large of a maximum learning rate will oscillate the weights too great and could lead to divergence. The authors proposed a learning rate test where the learning rate was slowly increased from one value to another for about

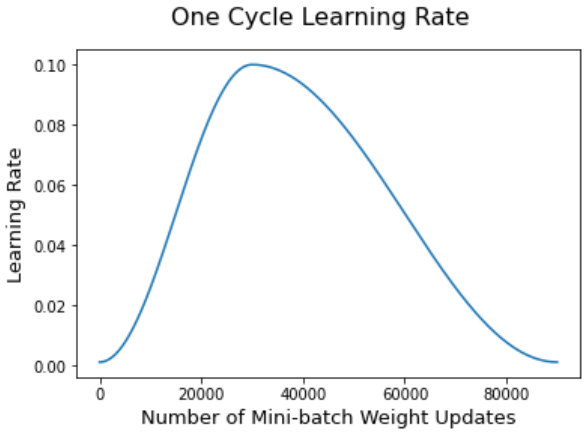



Figure 3.4: Single Cycle Learning Rate Schedule

10,000 minibatch weight updates. After the test, the maximum learning rate was selected to the learning rate right before any major oscillations or drops in performance. The authors showcased the power of this type of scheduling by achieving 92.4% on the CIFAR10 benchmark dataset within only 10,000 mini-batch updates using a Resnet56 model while cycling the learning rate from 0.1 to 3 on SGD optimizer. In comparison, their control group was Resnet56 model trained with a static learning rate of 0.35 on SGD, which achieved only 91.2% after 80,000 mini-batch updates. They termed this phenomenon super convergence.

# Chapter 4

# Approach and Methodology

In Chapter 4, Approach and Methodology, the contributions of this work will be explained in detail. As a reminder, the purpose of this work is to adapt the NASNet search space to include the optimization of non-convolutional operators, project the representation of the adapted NASNet search space into a static fixed length multi-dimensional vector so that vector optimization algorithms can be applied, and propose an algorithm, which was later found to be a genetic algorithm, with capabilities of maximizing exploration and exploitation given a limited number of fitness evaluations.

## 4.1 Changing the NASNet Search Space

The NASNet search space serves as a staple backbone from which many NAS systems can choose to augment for their optimization needs, as was seen with AmoebaNet. In this work, the NASNet search space was augmented for a few various reasons. First, the 7x7 separable convolution operator was removed as it is equivalent to two stacked 3x3 separable kernels, leading to a reduced cost in

both memory and computation for training a model with such a layer. Second, the NASNet search space was expanded as it lacks the ability to optimize non-convolutional operational layers, such as activation, batch normalization, and dropout. These non-convolutional operational layers were kept static, hand designed from the domain research in both NASNet and AmoebaNet by following up convolutional layers with batch normalization and ReLU activation. In this work, the search space was extended in order to include the optimization of these layers in order to explore more possible network architectures, while also breaking traditional ground. This new addition was compiled into a new building block, termed the After. The After, refers to the cleanup maintenance performed after the operations of a hidden state node. Once the chosen operations are applied to their respective inputs and are combined through element wise addition at a hidden state node, the output is run through the After. Figure 4.1 shows hidden state node 1 of an example architecture, where after filter concatenation is applied, the After block follows before sending off the input.

The After contains three layers: activation, batch normalization, and dropout. The choices available for optimization include selecting the activation function for the activation layer, which include ReLU, Leaky Relu, SeLU, and ELU; independent inclusion of the batch normalization and dropout layers; and lastly, the order in which the activation function and batch normalization was constructed. In order to see the possible orderings and inclusions, observe Figure 4.2. Figure 4.2 showcases two possible paths from the input that both then end with the dropout layer. One path is activation then batch normalization, and the other is batch normalization then activation. The green boxes refer to layers that are always included, while the tan boxes refer to layers that can be included.

By allowing the activation function to change for each independent node, the

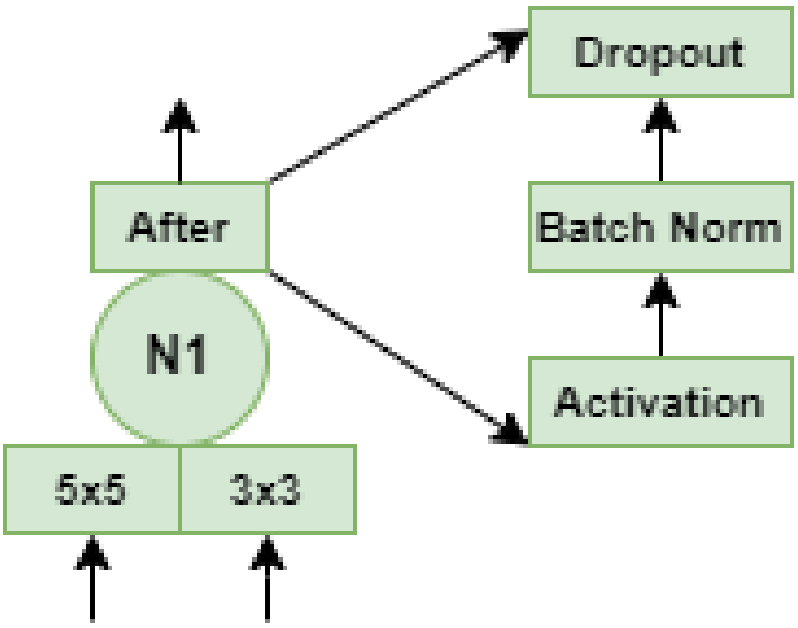



Figure 4.1: The After Block

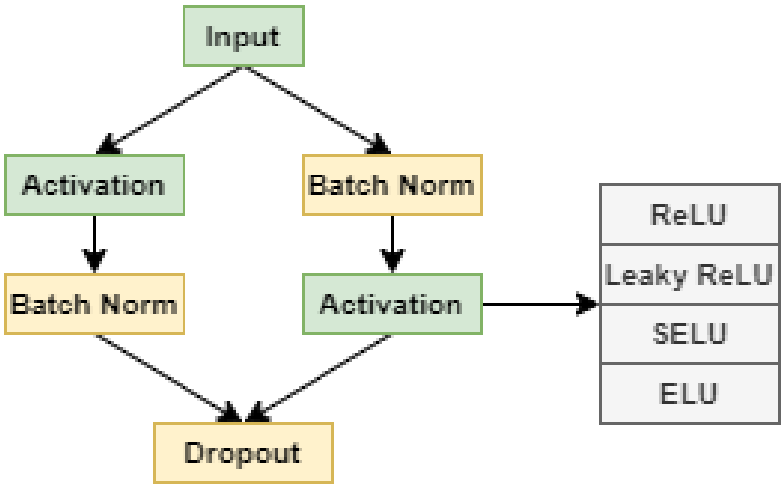



Figure 4.2: Order for After

search space now allows for architectures to select specific activations based upon the chosen operations at the given hidden node. Although batch normalization was originally applied before the activation [13], as it is currently debated whether

or not batch normalization solves the internal covariate shift phenomenon or acts as a reguralizer [23]. As a result, by incorporating a mechanism with the ability to select the order of batch normalization and activation, the search spaces allows to independently select the best option on a per hidden state node basis. In addition, the search space allows for the independent inclusion of batch normalization and dropout. Although batch normalization was created to solve the internal covariate shift, little theoretical basis supports its effectiveness in practice [8]. By allowing to independently include, or not, batch normalization at each hidden state node, the search algorithm has the ability to select which choice is best on a per node basis. In the original NASNet search space, the only dropout layer included was right before the softmax output layer. In this work, the search space was extended by allowing the inclusion, or not, of a dropout layer as the last layer of the After in order to include regularization or not for a particular node. The percentage dropout for the dropout layer is hand designed by the architect, allowing for scalability based upon the model at hand.

In addition to applying an independent After, after each hidden state node, each normal and reduction cell also had the ability to design an After for their output connection to the next cell and for their skip connection, which can be seen in Figure 4.3, where the After from the current cells filter concatenation is applied to two different connections, one for the next cell and the other for the skip connection. Lastly, an After was inserted right before the softmax activation layer, but after the global average pooling layer.

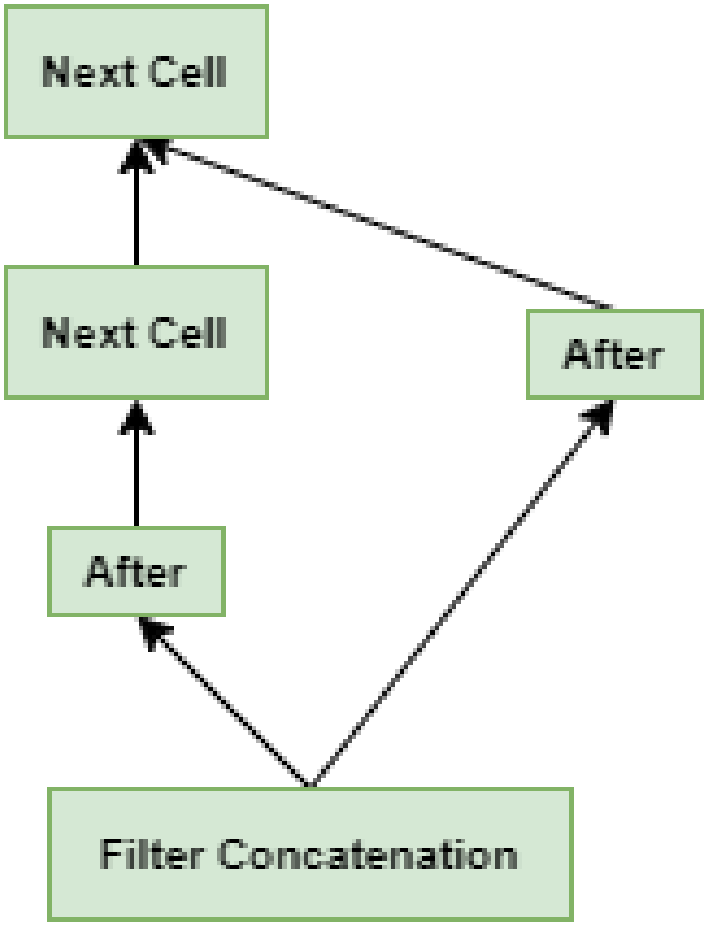



Figure 4.3: After for Cell

## 4.2 Constructing The Chromosome

The NASNet model creation has been defined as more of a phenotypical creation than genotypical. In the original work, the model was recursively created using a recurrent neural network controller by taking the argmax of the softmax output from a designated set of possible choices. In AmoebaNet, the model representation was more genotypical than NASNet, but still represented each model as a directed graph. The issue at hand becomes that common non-classical optimization algorithms such as particle swarm optimization, or low-level genetic

algorithms utilizing mean crossover or vector mutation, require fixed length chromosomes containing continuous floating-point values. In this work, a fixed length chromosome array with continuous floating-point values for the altered NASNet search space is proposed.

For each of the selected PSO and genetic algorithms, which will be discussed extensively later, each individual in the population was composed with a normal cell, a reduction cell, and output After. Each normal and reduction cell was composed of $n$ number of hidden state nodes, and a next connection After. Each hidden state node was composed of four choice vectors and an output After. The new proposed chromosome first starts at the hidden state node level. Algorithms 4, 5, and 6 detail the structure of the individual, normal and reduction cells, and hidden state nodes.

**Algorithm 4** Individual Object Structure

```
Object Individual
    normal_cell : Cell
    reduction_cell : Cell
    After : float[7]
    age : int                                  ▷ Used by GA and PSO
end Object
```

**Algorithm 5** Normal and Reduction Cell Object Structure

```
Object Cell
    hidden_nodes : HiddenNode[n]
    After : float[7]
end Object
```

**Algorithm 6** Hidden Node Object Structure

**Object** HiddenNode
*connection_vector_1* : FLOAT[*index*]
*connection_vector_2* : FLOAT[*index*]
*operation_vector_1* : FLOAT[12]
*operation_vector_2* : FLOAT[12]
$After$ : FLOAT[7]
**end Object**

At the node level, four choice vectors are utilized for constructing the chosen operations and connections. Two of the four are for selecting connections, one for each connection, and the rest of the two are for selecting operations, one for each operation. The two operation choice vectors are fixed length arrays containing 12 elements, one for each possible operation, where the selected operation is the argmax from the choice vector.

The two connection choice vectors are fixed length arrays containing *index* number of elements, where *index* refers to the hidden state node index in the cell. The chosen connection is calculated to be the argmax index of the choice vector. Each cell begins with two hidden state nodes, *index* 0 for the output from the previous cell and *index* 1 for the output from the skip connection. The second hidden state node *index* contains a connection vector with two elements, index zero for the previous cells connection and index one for the skip connection. The third node *index* now has a connection vector containing three elements, index zero for the previous connection, index one for the skip connection, and index two for the output from hidden node *index* 2. This process can be repeated ad infinitum, depending upon the number of chosen hidden nodes. Because the selected connection and operation are defined to the argmax of the respective choice vectors, the exact initialization does not matter; however, for clarity, all choice vectors were uniformly randomly initialized between -1 and 1. Figure 4.4

displays an example of this where three nodes are given, where beside each node are the two connection choice arrays already initialized with their random values. From node one, the argmax for both connection one and two is one, indicating that both op1 and op2 receive their input from the skip connection. Moving onto node two, the first connection array has an argmax of zero, indicating that op1 takes input from the previous connection, while op2 takes input from node one as the argmax for connection array two is two. The same process can be applied for node three.

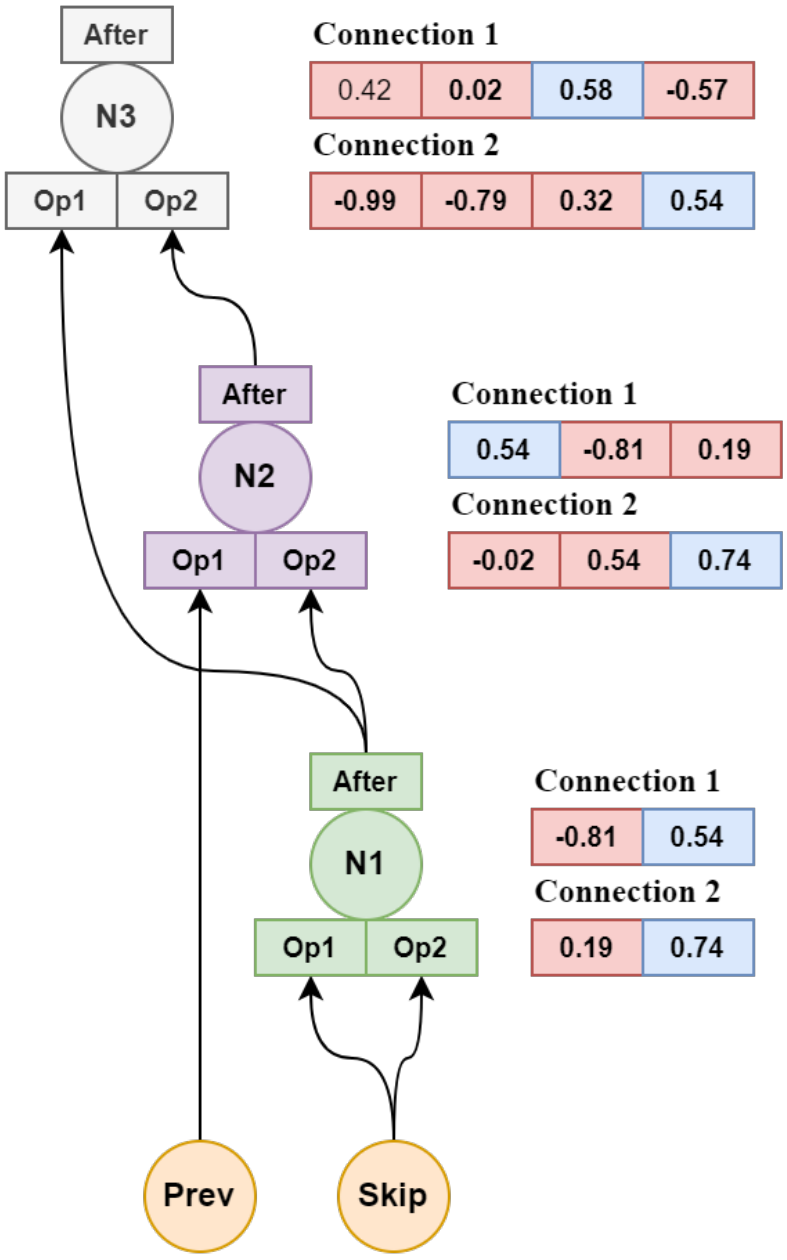



Figure 4.4: Example for Constructing Connections from Connection Choice Array

The chromosome for an After was defined to be a static seven element array uniformly randomly initialized between -1 and 1. If an element for an After was greater than zero, it was considered to be on, else it was considered to be off. If the zero-index element of the After array was on, then batch normalization was applied after activation, else it was before. If index five was on, then the batch normalization layer was included. If index six was on, then the dropout layer was included. Indices one through four dealt with the selection of the activation layer. Each of the four indices referred to the four possible activation functions, ReLU, Leaky ReLU, SELU, and ELU. Like the choice vectors previously described, the chosen activation function was the argmax of these four indices. In Figure 4.5, an example After array is given. Elements from the array that are red indicate that these elements either turned off a layer or were not the argmax from the activations. On the contrary, elements in blue indicate that these elements either turned on a layer or were the argmax from the activations. Because the element for order was positive, path one was utilized instead of path two. Because the element for ELU was the largest from the activation elements, the chosen activation was ELU. Because the element value for batch normalization was negative, batch normalization was turned off and not included. Lastly, because the element value for dropout was positive, dropout was turned on and included.

As it now has been described, each node can now be represented by a static floating-point dimensional vector as the only variable is dependent upon the number of hidden state nodes, which is statically defined by the architect. Each normal and reduction cell is composed of $n$ number of hidden state nodes, not counting the previous or skip connections, along with an After vector for the skip and next connections. Because the After vectors are static seven element arrays, and each of the $n$ hidden state nodes are static floating-point dimensional ar-

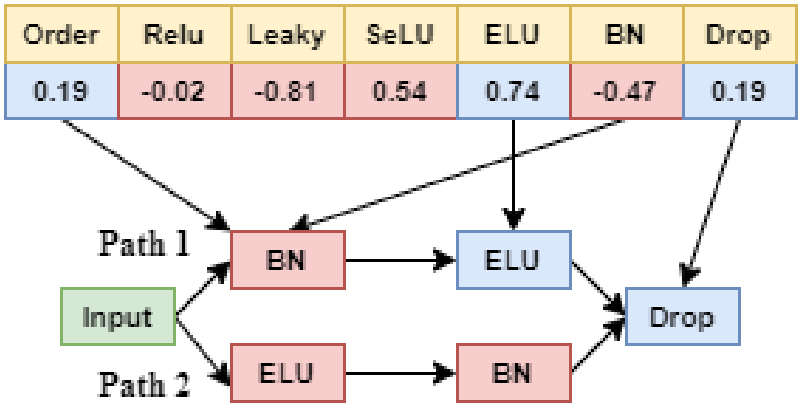



Figure 4.5: Example for Constructing After from After Array

rays, each normal and reduction cell are static floating-point dimensional arrays. Because each individual is composed of a normal and reduction cell, along with an After for right before the output layer, each individuals entire genotype can be represented by a static floating point dimensional array. Because each individuals genotype can be represented by a static floating point dimensional array, the doors have been opened for more non-classical optimization problems to be applied. The contribution of this new representation is exploratory as now any non-classical optimization technique requiring static floating-point dimensional arrays can be applied to the NASNet search space.

## 4.3 Constructing the Algorithms

In order to efficiently navigate the newly designed altered NASNet search space representation, two primary algorithms were compared and contrasted, a genetic and a particle swarm algorithm. One of the primary foci of this work is to efficiently search throughout the search space for network architectures while also reducing the massive amount of computation required. Previous NAS systems,

such as NASNet and AmoebaNet, explore and train 20,000 models, equating to thousands of GPU hours worth of computation. In this work, only 1,300 models will be trained and evaluated in order to showcase efficiency. It is expected that with such a limited number of total fitness function evaluations allowed, constructing the algorithms for convergence will greatly hinder exploration and may pre-convergence to poor solutions. As a result, the 1,300 fitness function evaluations were split into two separate sections, one for a global search and another for a local search. The proposed PSO and genetic algorithms operated with the goal in mind of a global search, while a mutation only genetic algorithm was utilized to refine the best solutions for a local search. The global search component utilized 1,000 fitness function evaluations and the local search utilized only 300 fitness function evaluations.

When constructing the algorithms, a number of concerns arose; specifically, the impact of the initial population size, dealing with the selection of hyper parameters for the GA, dealing with the selection of hyper parameters for the PSO, and the impact of model training on reliability of repeatable results.

### 4.3.1 Impact of Initial Population Size

The purpose of the initial population is to introduce a wide array of diversity from which the optimization algorithm can begin its optimization. It is common to uniformly randomly initialize the initial population from the search space at hand. However, the initial population size is often coupled with the size of the population for the algorithm. A smaller population size allows for convergence, but can lead to sup-optimal solutions, where each run can be drastically different due to the initialization. A larger population can allow for better exploration, but

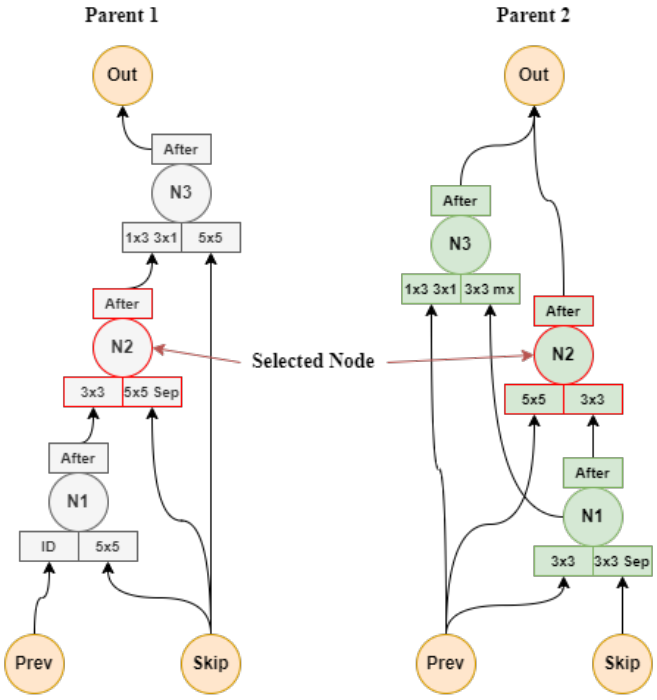
may take more iterations in order to achieve convergence. As a result, the initial population size can be seen as a variable that also controls the performance of the algorithm. In this work, the initial population size is detached from the population size during the loop of optimization. By detaching the initial population size, the initial population size can be increased to encourage initial diversity and then greedily cut down to the standard population size for convergence during the run of optimization.

### 4.3.2 The Genetic Algorithm

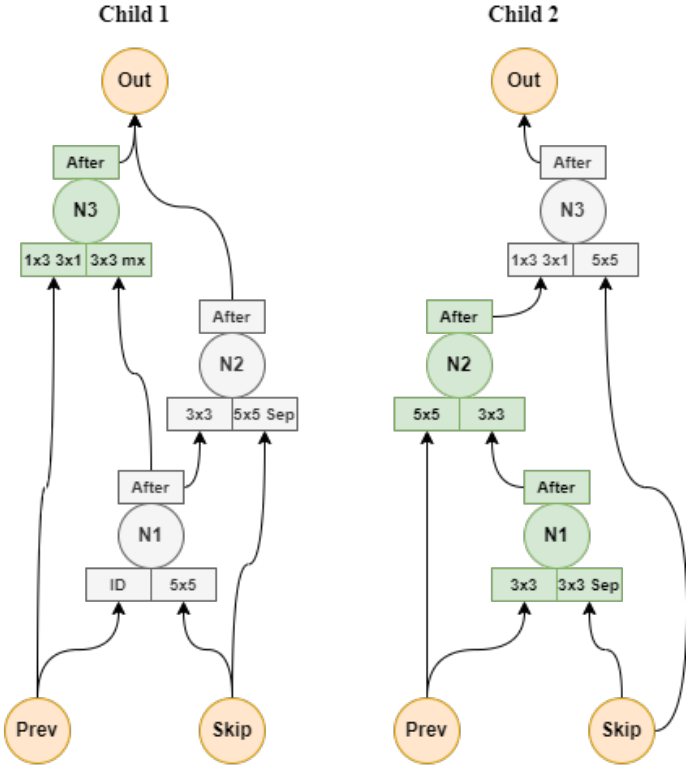
As detailed in section 2.4, genetic algorithms require a representation, selection operator, reproduction operator, survival operator, and selection for hyper parameters. As previously discussed in section 4.2, the chromosome representation for the genetic algorithm is a fixed length floating point multidimensional array. For the selection operator, random selection for pairing individuals from the population for mating was utilized in order to encourage exploration. To balance the random selection, elitism was utilized for the survival operator, in order to achieve convergence. Two primary forms of elitism were compared, parent level elitism and global level elitism. Parent level elitism refers to greedily taking the best set of individuals from the immediate parent offspring pairs to form the next population, while global level elitism greedily takes the best set of individuals from the entire pool of parent and offspring pairs. With the success of AmoebaNet in being able to utilize aging evolution as a source of regularization, aging was also implemented with the survival operator. However, instead of removing the oldest individuals from the populations with each iteration, a maximum age was set forth such that the children always replaced their parents if their parents age

exceeded a maximum set age limit. This was performed so that the information of the parents were not lost as the children can be extremely different due to crossover being a global search operator.

For reproduction, both crossover and mutation were utilized. Previously defined crossover and mutation operators in section 2.4.5 are performed with a focus at the genotype level. In contrast, the crossover and mutation operators proposed in this work are performed with the phenotype level in mind, but still operate at the genotype level. However, any of the previously defined crossover and mutation operators could be applied as the representation has been projected into a fixed length continuous array. Crossover was performed at the phenotype level by randomly selecting, from a parent, one of the $n$ hidden state nodes as a crossover point, and then swapping the sub tree above and below the crossover point with the other parent in order to create two children. Each hidden state node contains five components, two for choosing connection, two for choosing operations, and one for the After. All five components are swapped during crossover for a node. See Figure 4.6*a*, for an example where the second hidden state node was chosen as a crossover point when there exist only three hidden state nodes, and see Figure 4.6*b* for how the children were constructed from the crossover point. Despite crossover being applied from a phenotype level perspective, because cell is composed of nodes with fixed length arrays, this crossover mechanism can be seen at the genotype level of simply swapping subsections of the chromosome, similar to multi-point crossover. Crossover was applied for the both the normal and reduction cells of an individual, where the crossover point was independently randomly chosen for both. Algorithm 7 showcases the full crossover operator, where $random\_node()$ selects the random node index .




(a) Crossover Point




(b) Creation of Children

Figure 4.6: Crossover Operator Example

**Algorithm 7** Crossover Operator

**Input:** Parent_1 (type: Individual); Parent_2 (type: Individual)
normal_cell_crossover_point = random_node()
reduction_cell_crossover_point = random_node()

child_1 = copy(Parent_1)
child_2 = copy(Parent_2)

**for** *node* ***until*** *normal_cell_crossover_point* **do**
    `child_1.normal_cell.nodes[node] =`
        `Parent_2.normal_cell.nodes[node]`
**end**
**for** *node* ***until*** *reduction_cell_crossover_point* **do**
    `child_1.reduction_cell.nodes[node] =`
        `Parent_2.reduction_cell.nodes[node]`
**end**
**for** *node* ***until*** *normal_cell_crossover_point* **do**
    `child_2.normal_cell.nodes[node] =`
      `Parent_1.normal_cell.nodes[node]`
**end**
**for** *node* ***until*** *reduction_cell_crossover_point* **do**
    `child_2.reduction_cell.nodes[node] =`
        `Parent_1.reduction_cell.nodes[node]`
**end**
**Return** child_1, child_2

In this implementation, mutation was applied after crossover. At the phenotype level, mutation was implemented to be changing the connection, operator, or After of a particular node at three different levels. First, one of the hidden state nodes was randomly selected and either their connection, operator, or After was mutated. At the genotype level, mutation was implemented by randomly swapping the argmax value with another random indexed value from the choice array for either the connection, operator, and activation arrays. For other aspects of the After, mutation was implemented by changing the sign of the element in order to turn a layer from off to on or from on to off. Mutation was only applied once, to only one of these components, where the choice for which was chosen uniformly random. Second, mutation was applied to the After of the last layer of the cell with 25% probability. Third, mutation was applied for the After applied right before the output softmax layer. This last After, applied right before the output softmax layer, was never crossed over; but instead, was only mutated with 25% probability when creating the child. Algorithm 8 showcases the full mutation operator, where *normal_cell_mutation_point* selects the random node index of the normal cell to be mutated.

**Algorithm 8** Mutation Operator

**Input:** child (type: Individual)
normal_cell_mutation_point = random_node()
red_cell_mutation_point = random_node() ▷ red for Reduction

r = random.uniform(0, 1)
**if** *r > 0.66* **then**
  child = mutate_connection(
    child.normal_cell.nodes[normal_cell_mutation_point].connections)
**end**
**else if** *r > 0.33* **then**
  child = mutate_operation(
    child.normal_cell.nodes[normal_cell_mutation_point].operations)
**end**
**else**
  child = mutate_after(
    child.normal_cell.nodes[normal_cell_mutation_point].after)
**end**
**if** *if random.uniform(0, 1) < 0.25* **then**
  child = mutate_after(child.normal_cell.after)
**end**

▷ Reduction Cell

r = random.uniform(0, 1)
**if** *r > 0.66* **then**
  child = mutate_connection(
    child.reduction_cell.nodes[red_cell_mutation_point].connections)

**end**
**else if** *r > 0.33* **then**
  child = mutate_operation(
    child.reduction_cell.nodes[red_cell_mutation_point].operations)
**end**
**else**
  child = mutate_after(
    child.reduction_cell.nodes[red_cell_mutation_point].after)
**end**
**if** *if random.uniform(0, 1) < 0.25* **then**
  child = mutate_after(child.reduction_cell.after)
**end**
**if** *random.uniform(0, 1) < 0.25* **then**
  child = mutate_after(child.after)
**end**
**Return** child

The inclusion of the crossover operator was to efficiently globally search the domain space while mutation was utilized for local search. As stated in section 2.4.8, genetic algorithms have many different hyper parameters to set before evolution, such as probabilities of mutation and crossover. In order to reduce the need for tuning, this work did not utilize such probabilities; but instead, always performed crossover and mutation between a given set of parents, paired with either global or parent level elitism to ensure the information of the parents were not lost, as well as aging for a source of regularization.

From the crossover and mutation operators, the reproduction operator was built with the goal of creating the next generation. Algorithm 9 displays the overall process where the parents are randomly paired, and the pairs of parents begin reproduction. Crossover is applied to achieve the two children, which then undergo mutation. If parent elitism is enabled, then only the best two from the immediate parent offspring pair are selected for the next generation. If global elitism is enabled instead, then the best half from the total pooled parents and children are selected for the next generation. In addition, the $max\_age$ component is also utilized to ensure that those which are greater than the maximum age limit are always rejected. Together, the entirety of the genetic algorithm can be detailed in Algorithm 10.

**Algorithm 9** Reproduction Operator

**Input:** Generation (type: array of Individual); parent_elitism (type: Bool); global_elitism (type: Bool); max_age (type: int)

```
Parents = random_pairing(Generation)
next_generation = []

for (Parent_1, Parent_2) in Parents do
    child_1, child_2 = crossover_operator(Parent_1, Parent_2)
    child_1 = mutation_operator(child_1)
    child_2 = mutation_operator(child_2)
    if parent_elitism then
        next_generation.add(best_two(Parent_1, Parent_2, child_1,
        child_2, max_age))
    end
    else
        next_generation.add(Parent_1, Parent_2, child_1, child_2))
    end
end
if global_elitism then
    next_generation = best_half(next_generation, max_age)
end
Return next_generation
```

The last hyper parameter needing to be tuned was the population size. Three different sizes were tested, 10, 16, and 20. It was expected that the smaller population sizes would achieve better convergence while the larger would achieve better exploration. Convergence was defined to be when the models in the population become extremely similar to one another or when the mean fitness approaches the best fitness.

**Algorithm 10** Proposed Genetic Algorithm

**Input:** Fitness Function, $F : S_c \xrightarrow{\Phi} \Gamma \rightarrow \mathbb{R}$; max_gen (type: int); init_size (type: int); gen_size (type: int); max_age (type: int)

```
init_population = random_initialization(init_size)
init_fitness = fitness_function(population)
population, fitness = take_best(gen_size, init_population,
init_fitness)
for i until max_gen do
    increment_ages(population)
    population = reproduction_operator(population, parent_elitism,
    global_elitism, max_age)
    fitness = fitness_function(population)
end
```

**Return** Best Individual from Population

### Cascading Genetic Algorithm

Lastly, a cascading genetic algorithm was introduced. From the discussion on the influence of the initial population size, this was extended to an algorithm that starts off with a very large population size and slowly decreases it over the course of evolution to encourage exploration early on and exploitation later on. The exact rate of decrease could open up discussion, but was set to a step-wise schedule, starting with $n$ individuals in the population for three iterations and then reducing down by $\lfloor n/2 \rfloor$ every subsequent $4^i$ iterations, where $i$ begins at 1 and increases with each subsequent reduction, until a population size of 6 was reached. All together, the population reduction schedule followed the schedule given in Table 4.1, where a population size of 100 was utilized for 3 iterations, before switching to a population size of 50 for 4 iterations, and so on, until 1,000 fitness functions were reached for the global search phase.

Other than this reduction schedule, the cascading genetic algorithm still followed the same format as Algorithm 10.

Table 4.1: **Cascading Genetic Algorithm Reduction Schedule**

| Population Size | Number of Iterations |
|---|---|
| 100 | 3 |
| 50 | 4 |
| 25 | 8 |
| 12 | 16 |
| 6 | 19 |

### 4.3.3 The Particle Swarm Algorithm, with Subswarm

Two different PSO algorithms were utilized, the first being the gbest PSO algorithm, discussed in section 2.5.1, and a new PSO algorithm termed subswarm PSO. The gbest PSO algorithm performs very well at converging to solutions as each particle utilizes the best particles position in the calculation of the direction vector. On the other hand, the lbest PSO algorithm performs very well at hyperspaces containing many local minima, as each particle only utilizes the best particles position from a neighborhood of particles. However, when the number of fitness function evaluations is already limited to such a small degree, convergence can be considered more important. As a result, the gbest version of PSO was chosen in favor due to this reasoning.

To contrast gbest PSO, gbest subswarm PSO was also chosen. Subswarm optimization is an extension of PSO, also known as cooperative split PSO, first introduced by Van den Bergh and Engelbrecht [6]. The algorithm divides the population of particles into two or more subswarms that co-optimize different sections of a problem. In its first introduction, two subswarms of particles co-optimized both the weights and architecture of a neural network. The benefit of utilizing such a subswarm partition is that the algorithm is able to optimize sections of the problem instead of the problem at whole, which the authors argued

leads to finer grained searches. In the altered NASNet search space, there are two primary components that are optimized, the normal and reduction cell. Due to this independence, subswarm PSO can be applied in order co-optimize the normal and reduction cells by two independent swarms, one for normal and one for reduction. The structure and construction of both the velocity and position components of the particles are the same; however, each refer to a different subsection of the entire chromosome. As a result, evaluating each subswarm particle poses a challenge as each sub swarm particle only refers to a partial solution. The simple fix first provided by Engelbrecht was to take the best solution from the other swarm in order to construct the entire model for the current particle.

Both gbest and subswarm PSO have three hyper parameters, $w$ for inertia, $c_1$ for the cognitive component, and $c_2$ for the social component. The balance between these hyper parameters is vital in order to achieve exploration and exploitation. As a result, leaving these hyper parameters static can hinder both exploitation and exploration; however, implementing a reduction schedule where $c_1$ decreases while $c_2$ increases, to encourage exploration early on and convergence later on, can lead to premature convergence or no convergence if the rate is too high or low. As a result, a self-adaptive schedule is adopted based upon a cosine schedule [33]. The schedule works by adaptively changing the coefficients from a min value to max value following a cosine function. To do so, a related distance function was created to measure the distance between the fitness value of the current particle and the global best. By using this related distance function, the hyper parameters can self-adapt by increasing exploration, increasing $w$, if the distance is small, while increasing convergence, increasing $c_2$, if the distance is large. As a result, $c_1$ is made static, 2.05 in this implementation. The distance function is seen in Equation 4.1, where $f$ is the fitness value, $i$ is the $i^{th}$ particle,

$t$ is the current time step, $G$ is the global best.

$$\zeta_i^t = \frac{f_i^{t-1} - f_G^{t-1}}{f_G^{t-1}} \tag{4.1}$$

In order to achieve the cosine slope, two intermediary functions were created, one for $w$ and one for $c_2$. Equation 4.2 shows the new calculation for $w$, given the distance function value for the $i^{th}$ particle. Equation 4.3 shows the new calculation for $c_2$, given the distance function value for the $i^{th}$ particle.

$$w_i^t = 0.9 * 2(1 - \cos(\frac{\pi}{2}\zeta_i^t)) + 0.45 \tag{4.2}$$

$$c_{2_i}^t = 0.5 * 2.2(1 - \cos(\frac{\pi}{2}\zeta_i^t)) + 2.5 \tag{4.3}$$

Together, equation 2.11 and 2.12 for calculating the velocities and new positions of the particles can be updated to incorporate this new adaptive schedule, as shown in equations 4.4 and 4.5.

$$v_i(t+1) = w^t v_i(t) + 2.05 r_1(y_i(t) - x_i(t)) + c_2^t r_2(\hat{y}_i(t) - x_i(t)) \tag{4.4}$$

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{4.5}$$

Lastly, the concept of aging was also applied to both gbest PSO and subswarm to allow for the regularization of the global best position. To prevent holding onto a global best position that cannot repeat its fitness score, as the fitness function is stochastic, once the global best position has not changed for $age$ iterations, it was replaced by the global best from the current set of particles. This can be

seen in Algorithm 11 where in the *best_particle* update step, it incorporates the maximum age for selecting the new global best position.

**Algorithm 11** SubSwarm PSO Algorithm

**Input:** Fitness Function, $F : S_c \xrightarrow{\Phi} \Gamma \rightarrow \mathbb{R}$; max_iter (type: int); max_age (type: int)

```
swarm_1_particles = random_initialization()
swarm_1_velocities = random_initialization()
swarm_1_gbest_age = 0
swarm_2_particles = random_initialization()
swarm_2_velocities = random_initialization()
swarm_2_gbest_age = 0

for i until max_iter do
    for particle in swarm_1_particles do
        swarm_1_fitness.add(fitness_function(particle,
        best_particle(swarm_2_particles)))
    end
    swarm_1_gbest_position = best_particle(swarm_1_particles,
    swarm_1_gbest_position, swarm_1_gbest_age, max_age )
    increment swarm_1_gbest_age
    for particle in swarm_1_particles do
        update velocity using equation 4.4
        update position using equation 4.5
    end
    for particle in swarm_2_particles do
        swarm_2_fitness.add(fitness_function(particle,
        best_particle(swarm_1_particles)))
    end
    swarm_2_gbest_position = best_particle(swarm_2_particles,
    swarm_2_gbest_position, swarm_2_gbest_age, max_age )
    increment swarm_2_gbest_age
    for particle in swarm_2_particles do
        update velocity using equation 4.4
        update position using equation 4.5
    end
end
Return Best Particle from Each Swarm
```

### 4.3.4 Training the Models

Despite reducing the total number of models allowed for creation, the reality of training and evaluating the models on a benchmark dataset still stands. Training a full scale model until convergence would be ideal, but the cost of training the model far outweighs the benefit as an extreme amount of computation would be necessary. Instead, each model was projected to $n = 2$, referring to only utilizing two cells in each normal cell module for creation, and only 3 hidden state nodes were used for each cell, instead of 5 as in NASNet and AmoebaNet. Although utilizing 3 hidden nodes may lead to a degrade in model performance, it speeds up training by providing a less complicated model and less number of parameters. The new proposed NASNet search space also allows for the inclusion of a dropout layer, where the dropout percentages were set based upon the model size. Each model was created by stacking a normal cell module with a reduction cell twice, and then ended with another normal cell right before the global averaging and softmax output, as shown in Figure 3.1. With this construction, there are four dropout percentages to be defined, one for each of the three normal cell modules and one for the last After before.

In addition to this smaller projection of the model, each model was only trained upon half of the selected benchmark dataset. By only using half the dataset for training, convergence can be expedited as the dataset difficulty has been reduced. The issue of over fitting could be an issue when using only half the dataset for training, but due to the model sizes already being extremely small projections, over fitting was not considered to be a problem. Lastly, in order to achieve reliable results, all models were trained until convergence on this smaller dataset. Convergence can take a many number of mini-batch weight updates

utilizing a standard learning rate; therefore, in order to reduce the number of weight updates, a one-shot cycle learning rate was utilized in order to speed up convergence. It is common for NAS systems to utilize shared weights when training models; however, this was not performed as it can be hard to distinguish between which the optimization algorithm is optimizing, the architecture or weights. To prevent over fitting the smaller model architecture by prioritizing the weights instead of the architecture, shared weights were not utilized and each model was trained from scratch. Together, achieving efficient and reliable model evaluation was discussed to be training extremely small scale models on a subsection of the benchmark dataset until convergence, while utilizing a one-shot cycle learning rate schedule.

# Chapter 5

# Experiments and Results

Section 4.3.2 introduced a few variants of the proposed genetic algorithm. First, either global or parent level elitism was able to applied as the survival operator. Second, aging could also be applied as a secondary level survival operator. Third, three possible generation sizes were proposed: 10, 16, and 20. Lastly, a cascading genetic algorithm variant was proposed, which reduced the generation size gradually to encourage exploitation. With the discussion on the importance of the initial population, all non cascading genetic algorithm variants began with an initial population size of 50, and greedily cut down to their respective generation sizes. Because only 1,000 fitness functions were available for the primary global search, a generation size of 10 ran for 95 iterations, 60 iterations for size 16, and 47 iterations for size 20. For the aging variants, the maximum age was set dependent upon the generation size. It was expected that algorithms with larger generation sizes need a longer threshold for age as they naturally take longer to converge. With this information, all possible genetic algorithm variants are detailed in Table 5.1.

Section 4.3.3 introduced a few variants of the proposed PSO algorithm. First,

Table 5.1: **Genetic Algorithm Variants**

| **Algorithm** | **Parent or Global Elitism** | **Population Size** | **Number of Iterations** | **Maximum Age** |
|---|---|---|---|---|
| GA | Parent | 10 | 95 | 5 |
| GA | Parent | 16 | 60 | 6 |
| GA | Parent | 20 | 47 | 7 |
| GA | Global | 10 | 95 | 5 |
| GA | Global | 16 | 60 | 6 |
| GA | Global | 20 | 47 | 7 |
| GA | Parent | 10 | 95 | None |
| GA | Parent | 16 | 60 | None |
| GA | Parent | 20 | 47 | None |
| GA | Global | 10 | 95 | None |
| GA | Global | 16 | 60 | None |
| GA | Global | 20 | 47 | None |
| CASC | Parent | - | - | 6 |
| CASC | Global | - | - | 6 |
| CASC | Parent | - | - | None |
| CASC | Global | - | - | None |

gbest was discussed in favor of lbest due to its property of accelerated convergence. Second, a gbest variant known as subswarm optimization was proposed as a means of optimizing the normal and reduction cells independently. Lastly, three possible population sizes were proposed: 10, 16, and 20. Like the genetic algorithms, all PSO algorithms began with an initial population of 50, greedily taking the best down to the size of the set population. In addition, the number of iterations and selected maximum ages per algorithm was the same as the genetic algorithm, dependent upon the population size. With this information, all possible PSO variants are detailed in table 5.2.

From tables 5.1 and 5.2, there exist 26 total possible algorithms proposed as viable algorithms to search the new NASNet search space representation. Unfortunately, the computational resources available were insufficient to test each

Table 5.2: **Particle Swarm Variants**

| Algorithm | Population Size | Number of Iterations | Maximum Age |
|---|---|---|---|
| PSO | 10 | 95 | 5 |
| PSO | 16 | 60 | 6 |
| PSO | 20 | 47 | 7 |
| PSO | 10 | 95 | None |
| PSO | 16 | 60 | None |
| PSO | 20 | 47 | None |
| SUBSWM | 10 | 95 | 5 |
| SUBSWM | 16 | 60 | 6 |
| SUBSWM | 20 | 47 | 7 |
| SUBSWM | 10 | 95 | None |
| SUBSWM | 16 | 60 | None |
| SUBSWM | 20 | 47 | None |

possible combination. Instead, a preliminary evolution was performed with each combination on a smaller gray scale version of the CIFAR10 dataset to obtain the best variant. Once the best combination was obtained, it was applied full scale to CIFAR10 for 1,000 fitness function evaluations for the global search. After optimization, a mutation only genetic algorithm was utilized for the local search phase by being applied to the final population of the algorithm in order to fine tune the best model. Finally, the best model was selected and projected for full scale testing on CIFAR10 and was then transferred to another closely related benchmark dataset, CIFAR100. CIFAR100 is benchmark dataset which contains 60,000 32x32 RGB images spanning over 100 unique classes.

## 5.1 Initial Algorithm Comparison

With 26 possible algorithms, a preliminary optimization was utilized in order to obtain the best combination for the full scale problem. This preliminary problem

was created to be smaller and easier than the actual CIFAR10 dataset in order to test the full optimization of each algorithm without wasting too many computational resources. The CIFAR10 dataset was reduced down from 10 classes to 2, leaving only 10,000 for training and 2,000 for validation. This was performed in order speed up convergence as the problem becomes easier to classify. The images were also projected to gray scale in order to reduce the dimensionality, to further reduce computation. In addition to the reduced dataset, as a means to create smaller models for faster evaluation, each model was only projected to $n = 2$ for number of cells per normal cell module, with 2 filters for the starting cell, and $n = 3$ for number of hidden state nodes. The chosen number of filters per module along with the dropout percentages are given in Table 5.3. These dropout percentages were not empirically decided but were through intuition based upon the model size.

Table 5.3: **Gray Scale CIFAR10 Model Number of Filters and Dropout Percentages**

| | Cell Module 1 | Cell Module 2 | Cell Module 3 | Last After |
|---|---|---|---|---|
| Number of Filters | 2 | 4 | 8 | - |
| Dropout Percentage | 2.5% | 5.0% | 7.5% | 10% |

Each model was only allowed to train for 14 epochs with batch size of 100, equating to only 1,400 mini-batch weight updates. Given the limited number of weight updates, a one-shot cycle learning rate schedule was utilized to achieve super convergence. The schedular had a minimum learning rate of 0.001 and a maximum learning rate of 0.1. The Adam optimizer was chosen for optimization as it utilizes momentum along with adaptive learning rates for each individual parameter for faster and more stable convergence.

Due to the very small dataset, using validation accuracy as the primary metric

during optimization was insufficient as many models often achieved the exact same validation accuracy; instead, a scaled validation loss was utilized as the primary metric as it was more unique. Cross-entropy loss is a metric that is to be minimized; therefore, to allow it to be primary metric for optimization by the proposed algorithms, it was scaled by $\frac{1}{loss}$ so that smaller loss values would yield higher fitness scores, as the goal is maximization. Each possible algorithm was ran three times to obtain a mean evaluation. Within each run, the best and min fitness were recorded with each iteration to track progress and convergence. In addition, the mean similarity between individuals was also tracked with each iteration to measure convergence as well. Similarity was defined to be the proportion of matching operations and connections between any two individuals.

### 5.1.1 Cascading Genetic Algorithm

The results for the cascading genetic algorithm are depicted in Figure 5.1. Figure 5.1 showcases aging vs. non aging for the algorithm, where the best fitness are depicted by the solid lines, while the min fitness are depicted by the dashed lines. As one can see, using global elitism along with no aging achieved the largest final best fitness as well as the largest final min fitness while having the lowest mean similarity. From this figure it appears that for the cascading algorithm, utilizing no aging performed better than the aging counterparts, where global elitism performed better than parent elitism in terms of achieving a better best and min fitness score. It is interesting to note that by incorporating aging, the mean similarities were higher than their non aging counterparts, which seem counter-intuitive as aging can be seen as a form of regularization against

convergence.

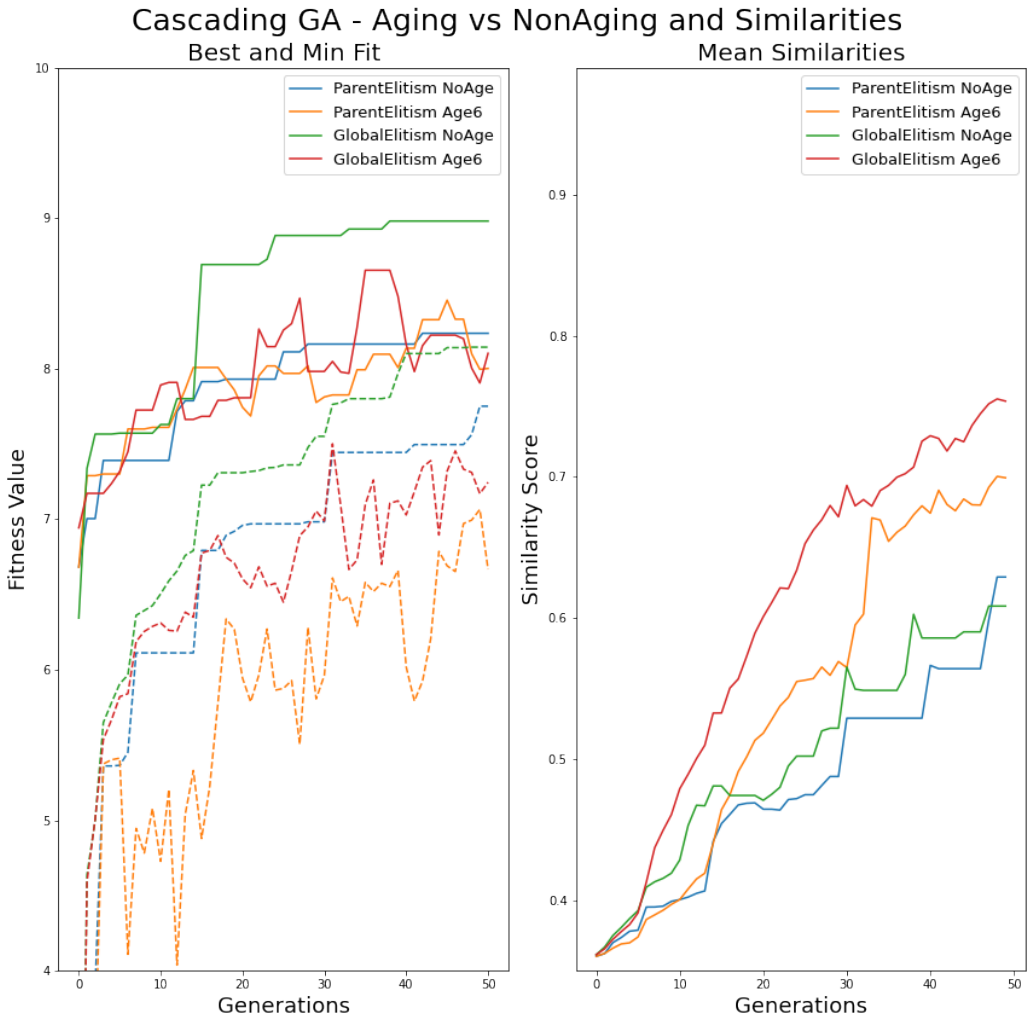



Figure 5.1: Gray-scale CIFAR10: Cascading GA Aging vs Non Aging

### 5.1.2 Aging Genetic Algorithm

The results for the aging genetic algorithm are depicted in Figure 5.2, where the best fitness are depicted by the solid lines, while the min fitness are depicted by the dashed lines. As one can see, using parent elitism with a population size of 20 achieved the largest final best and min scores out of the algorithms, while being middle of the pack for similarities. From this figure it appears that for aging algorithm, utilizing parent elitism performed better than the global elitism counterparts in terms of achieving a better best and min fitness score. As to be expected, the algorithms with smaller population sizes and longer iterations achieved more convergence as their mean similarities were higher than their counterparts.

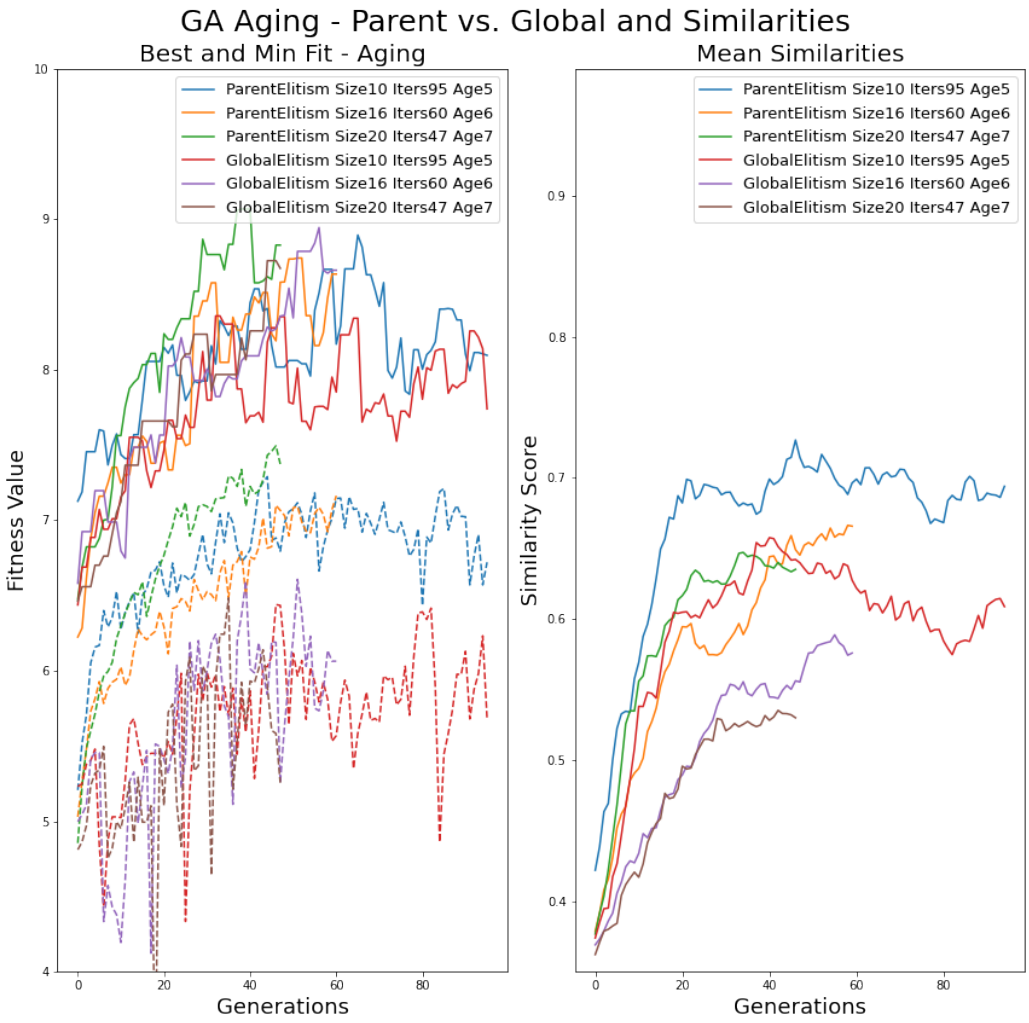



Figure 5.2: Gray-scale CIFAR10: Aging GA

### 5.1.3 Non-Aging Genetic Algorithm

The Results for the non aging genetic algorithm are depicted in Figure 5.3, where the best fitness are depicted by the solid lines, while the min fitness are depicted by the dashed lines. As one can see, using global elitism with a population size of 16 achieved within the top two for best fitness; however, it achieved the best minimum value by far when compared to the other algorithms, showcasing its ability to converge the population. For both population sizes 10 and 20, parent elitism performed better than their global elitism counterparts. As to be expected, the algorithms with smaller population sizes and longer iterations achieved more convergence as their mean similarities were higher than their counterparts.

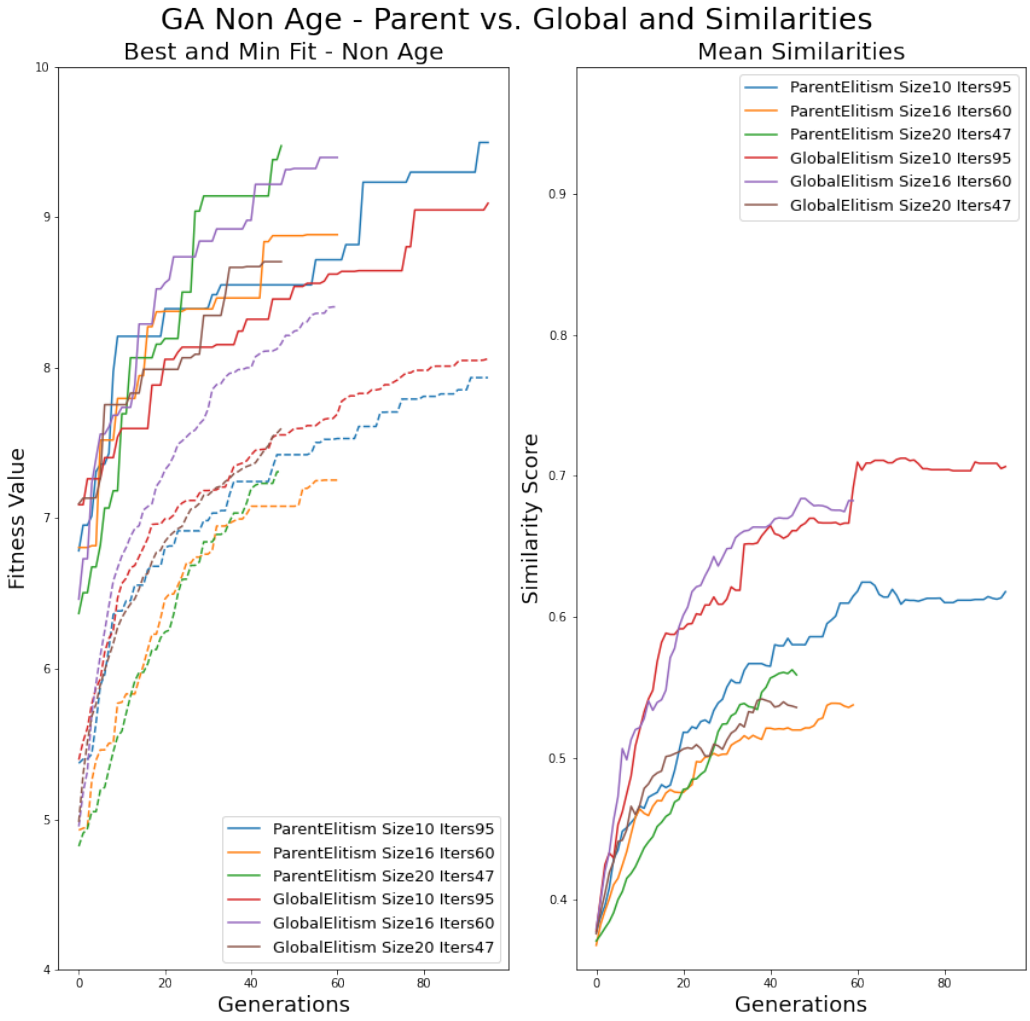



Figure 5.3: Gray-scale CIFAR10: Non Aging GA

### 5.1.4 PSO Algorithms

The results for aging vs. non aging PSO algorithm are depicted in Figure 5.4, where the best fitness are depicted by the solid lines, while the min fitness are depicted by the dashed lines. As one can see, using no aging with a population size of 10 achieved the largest best and min fitness values. From this figure, it appears that the non aging variants had better stable min fitness values, showcasing better convergence. It is interesting to note that the similarity scores are extremely large compared to the previous genetic algorithms, showcasing the ability for PSO to converge to solutions extremely fast.

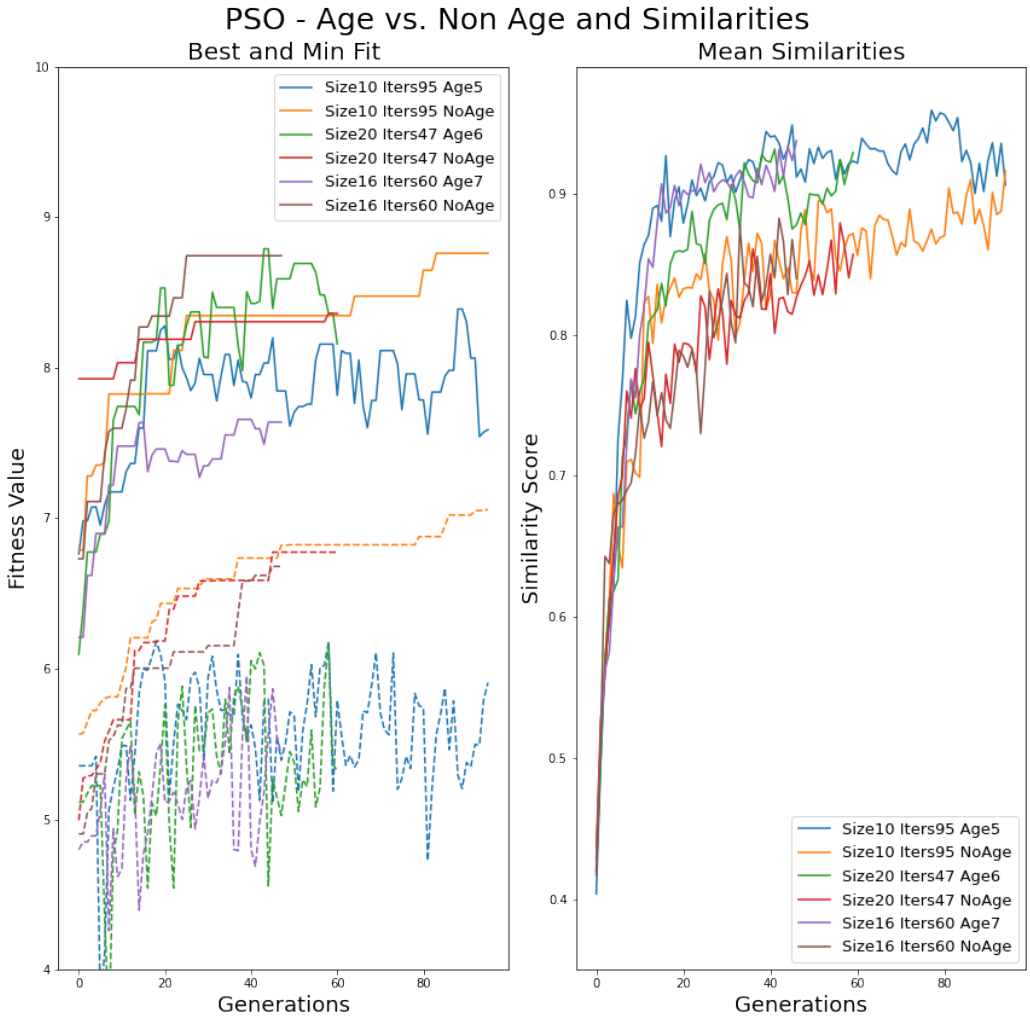



Figure 5.4: Gray-scale CIFAR10: Age vs. Non Age PSO

### 5.1.5 Subswarm PSO

Lastly, the results for subswarm PSO are depicted in Figure 5.5, where the best fitness are depicted by the solid lines, while the min fitness are depicted by the dashed lines. Similarity scores were not tracked as there were two independent swarms. As one can see, using no aging with a population size of 10 achieved the second largest best and largest min fitness values. It is interesting to note that the best fitness for population 20 with aging achieved the largest best score by far, but its min fitness did not follow.

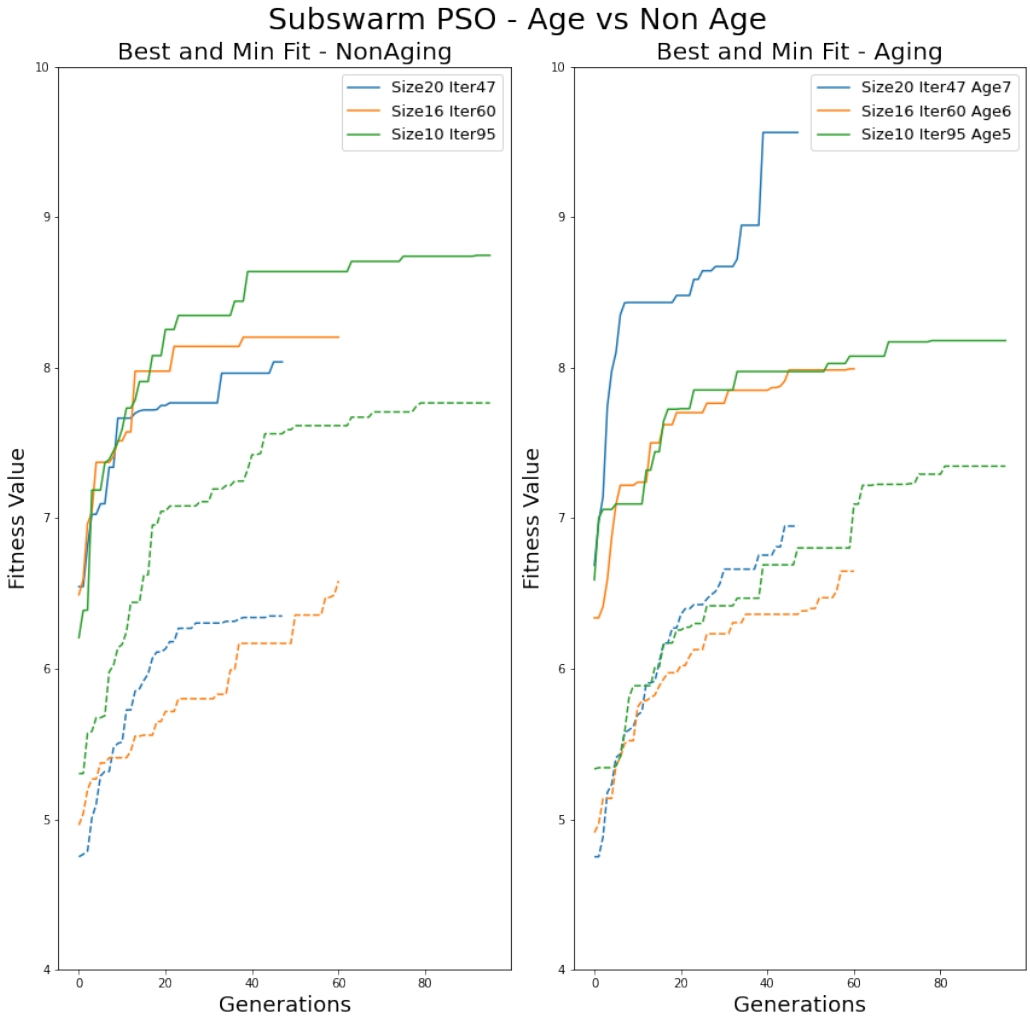



Figure 5.5: Gray-scale CIFAR10: SubSwarm PSO

### 5.1.6 Best Algorithms

Now that all the possible algorithms have been explored, the best from each of the previously discussed figures were compiled in Figure 5.6. Figure 5.6 showcases the best algorithms from each section in order to solidify on the final algorithm. As one can see, using a genetic algorithm with global elitism at a population size of 16 achieved the largest best and min fitness across all algorithms while also having above average convergence in terms of similarity scores. From these results, this algorithm was chosen as the final to be tested on the full scale evolution.

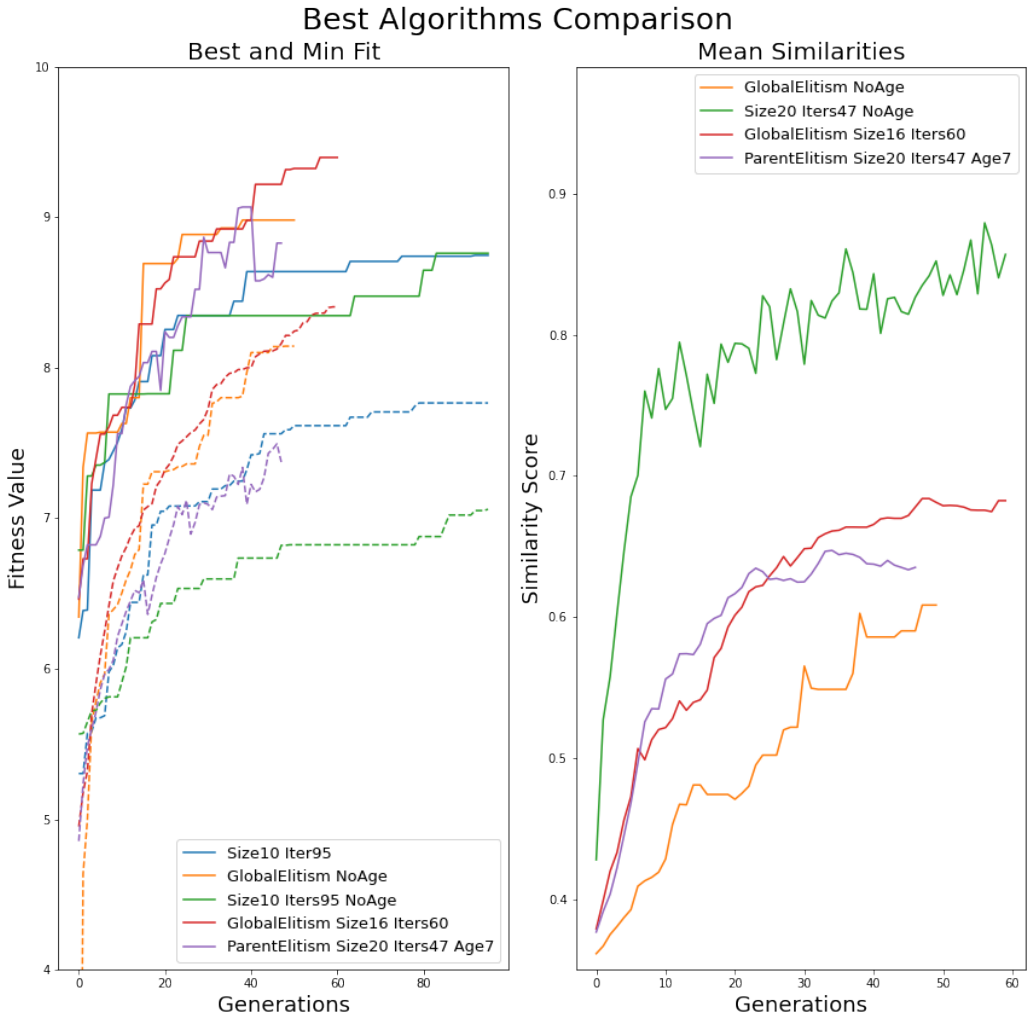



Figure 5.6: Gray-scale CIFAR10: Best Algorithms Comparison

### 5.1.7 Creation of Mutation Only Algorithm

As stated before, all 1,300 fitness function evaluations were split in between the global and local search phases, 1,000 for the global and 300 for the local. All of the previously discussed algorithms were applied to the global search phase. For the local search phase, the goal in mind is to narrow down the global search results to a local search about the best solution. Although a PSO algorithm could have been utilized due to its measured ability to converge to solutions, as the results above showcase, a mutation only algorithm was utilized in order to be consistent with using a genetic algorithm and previously defined research utilizing a mutation only genetic algorithm. The final population from the global search phase was utilized as the initial population for the mutation only algorithm. From the results discussed above, the cascading genetic algorithm achieved achieved the best convergence out of the genetic algorithms in terms of yielding a larger mean similarity score over evolution. Because the goal of the local search phase is to perform a local search, convergence is important. As a result, a cascading mutation only genetic algorithm was utilized to narrow down the final generation from the global search phase to a best solution. The population reduction scheme followed that given in Table 5.4.

Table 5.4: **Cascading Mutation Algorithm Reduction Schedule**

| Population Size | Number of Iterations |
|---|---|
| 10 | 5 |
| 5 | 5 |
| 3 | 8 |

Due to the success of non aging and global level elitism, the cascading mutation only genetic algorithm did not utilize aging and incorporated global level elitism. Reproduction was performed by creating three children from randomly

mutating one of the nodes of the parent. The cascading mutation only genetic algorithm still follows the format of Algorithm 10, but no aging component was utilized and the *reproduction_operator* was replaced with a specifically created reproduction operator for the mutation only algorithm. This new operator is detailed in Algorithm 12, which showcases that each individual creates three randomly mutated offspring, that are then pooled together with their parents for global level elitism.

**Algorithm 12** Cascading Mutation only Reproduction Operator

**Input:** Generation (type: array of Individual); gen_size (type: int)

```
for individual in Generation do
    next_generation.add(individual)
    next_generation.add(mutation_operator(individual))
    next_generation.add(mutation_operator(individual))
    next_generation.add(mutation_operator(individual))
end
next_generation = best_individuals(next_generation, gen_size)
Return next_generation
```

## 5.2 Evolution of Chosen Algorithms

### 5.2.1 Global Search Phase

The previous section compared and contrasted all of the proposed algorithms on a preliminary gray scale reduced CIFAR10 dataset. The best combination was found to be a genetic algorithm utilizing global elitism with an initial population size of 50 and generation size of 16. This algorithm was then projected to the full scale evolution on the entire CIFAR10 dataset. As mentioned in section 4.3.4, in order to reduce the computation required to train each model, the dataset size was reduced to half, while each model was projected to $n = 2$ for number of cells

per normal cell module, with 4 filters for the starting cell, and $n = 3$ for number of hidden state nodes. The chosen number of filters per normal cell module along with their dropout percentages are given in Table 5.5.

Table 5.5: **Full Scale Model: Number of Filters and Dropout Percentages**

| | Cell Module 1 | Cell Module 2 | Cell Module 3 | Last After |
|---|---|---|---|---|
| Number of Filters | 4 | 8 | 16 | - |
| Dropout Percentage | 5% | 10% | 15% | 20% |

From the 50,000 training images for CIFAR10, the first 20,000 were used for training the models while the last 10,000 were used for validation. The test images were not touched at all during any stage of evolution in order to prevent over fitting the CIFAR10 dataset as a whole. Each model was only allowed to train for 300 epochs with a batch size of 1000, equating to 6,000 mini-batch weight updates. In order to speed up convergence, a one-shot cycle learning rate was utilized, ranging from a min learning rate of 0.001 to 0.05. AdamW optimizer was utilized with a weight decay coefficient of $1e-7$. Unlike the gray-scale experiments, validation accuracy was utilized as the primary metric to optimize.

The global search phase evolution process was performed on one A100 GPU, elapsing 121 hours, equating to only 5 days, to evaluate 1,000 models. To begin, Figure 5.7 showcases the box plot of the initial population for the genetic algorithm, the validation accuracies range from 55% to 79% with median around 75%.

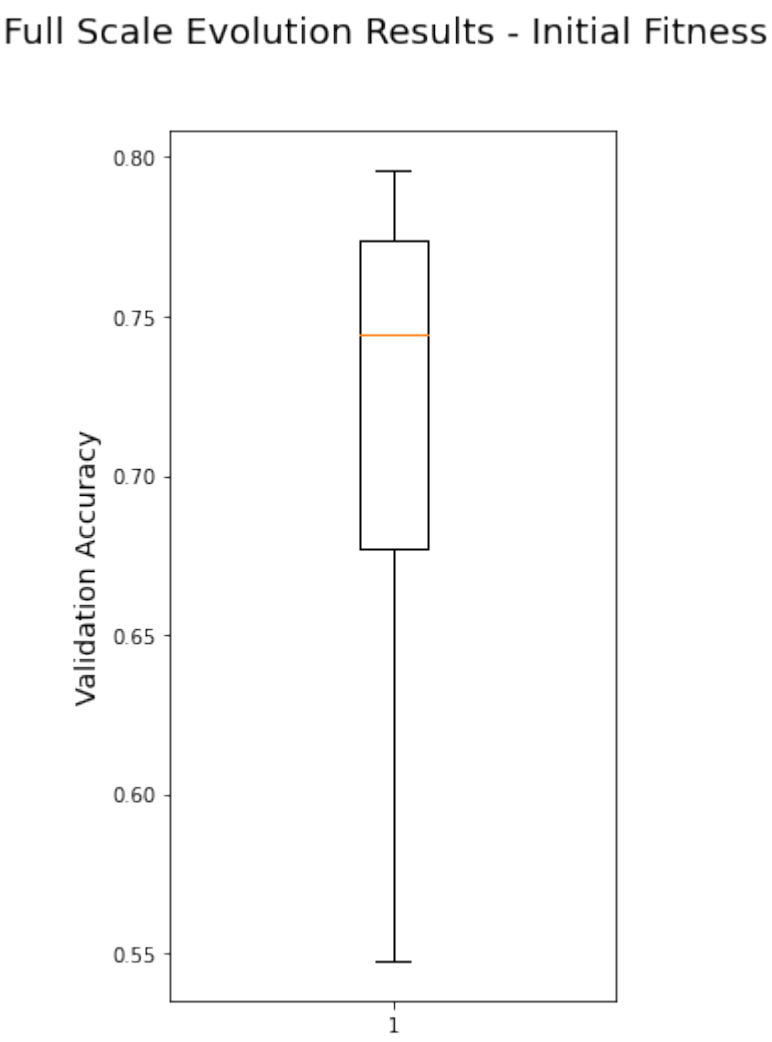



Figure 5.7: Full Scale Evolution Initial Fitness Results

The fitness results from evolution are depicted in Figure 5.8.

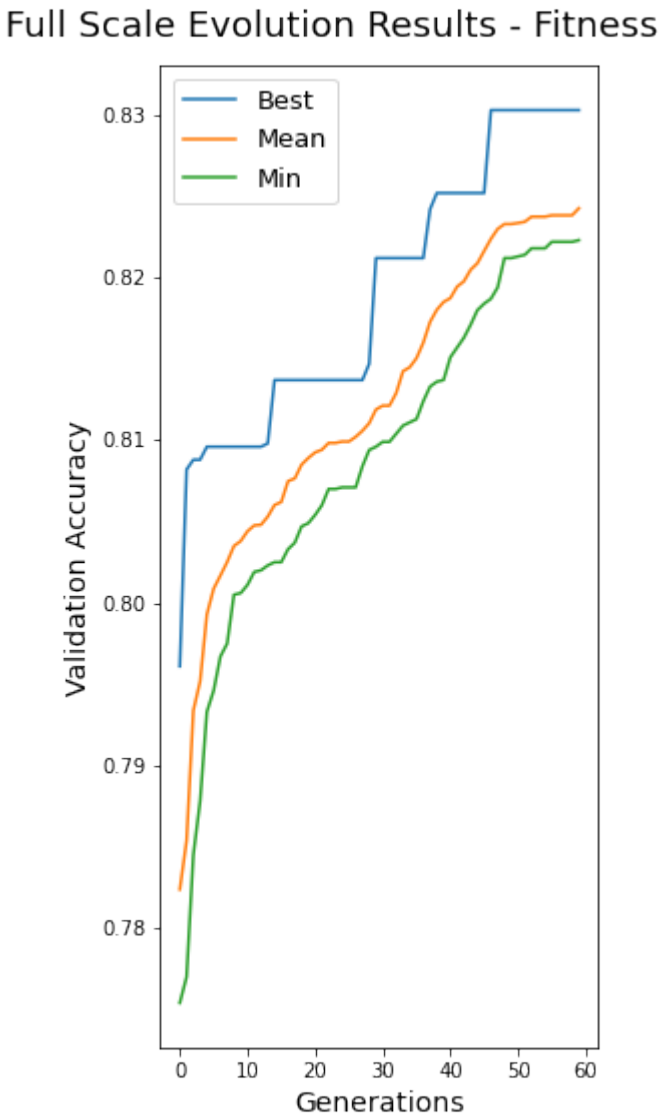



Figure 5.8: Full Scale Evolution Fitness Results

The similarity scores can be seen in Figure 5.9. The max and mean ages for each iteration from the evolution are shown Figure 5.10.

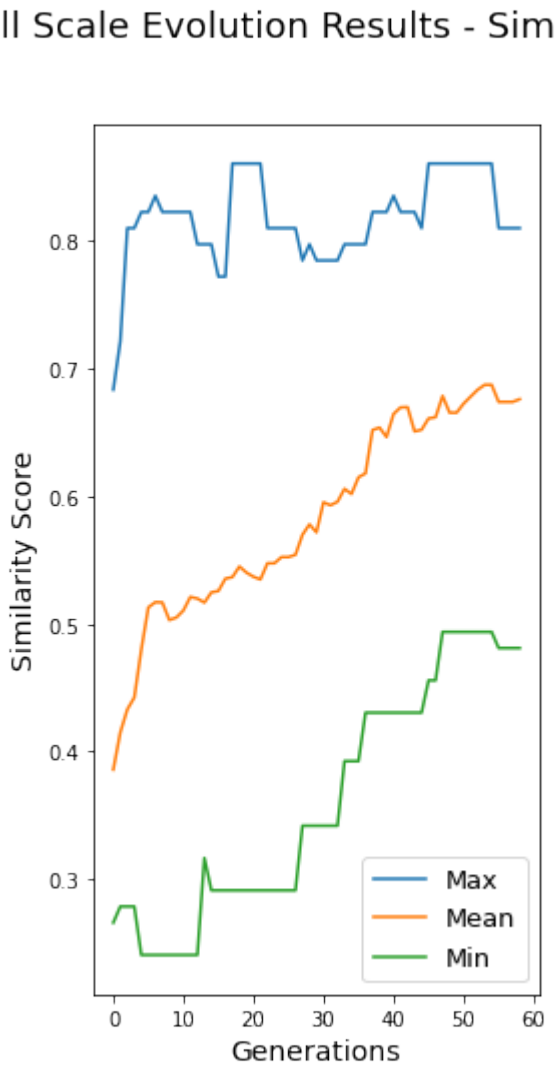



Figure 5.9: Full Scale Evolution Similarity Results

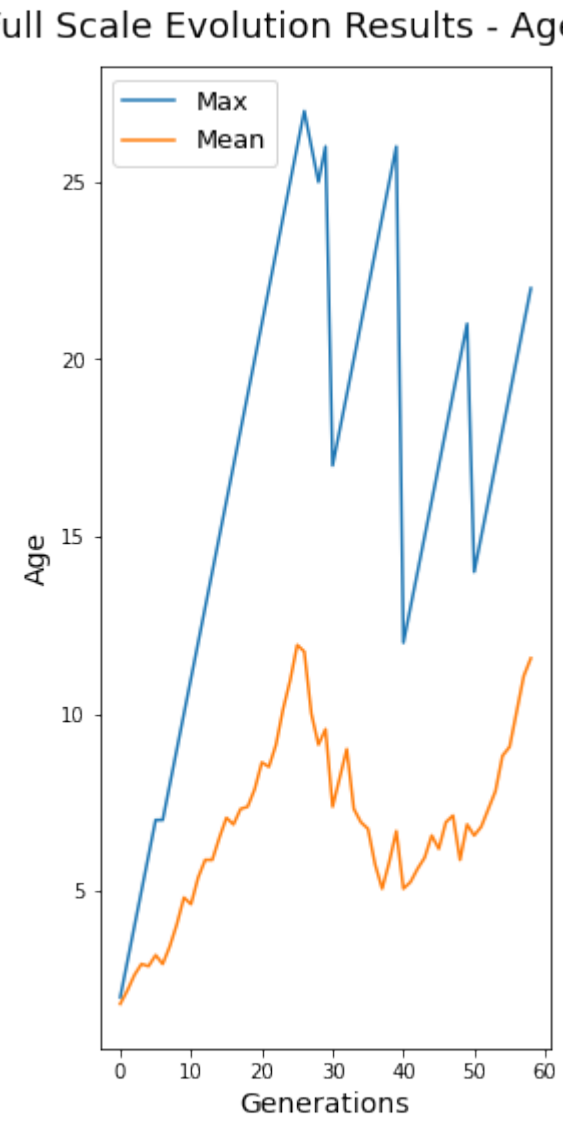



Figure 5.10: Full Scale Evolution Max and Mean Age Results

### 5.2.2 Local Search Phase

After the completion of the global search phase, the local search phase was enacted. Unlike the global search phase, the local search phase trained models with different parameters. Each model was projected to the same dimension as previously described in Table 5.5, but now each model was trained upon the entire data set instead of half, in order to get a better evaluation. From the 50,000 training images available for the CIFAR10 dataset, the first 40,000 were utilized to train the model and the last 10,000 were utilized as validation. Each model was trained upon this dataset partition for 200 epochs with batch size of 1,000, equating to 8,000 mini-batch updates, 33% more than the global search phase. Again AdamW, with a weight decay of $1e-7$, was utilized as the optimizer, along with a one-shot cycle learning rate schedule with an initial learning rate of 0.001 and a maximum learning rate schedule of 0.1.

The final generation from the genetic algorithm was trained using this new evaluation strategy and the best 10 were selected to be the initial population for the cascading mutation only genetic algorithm. The local search phase process was performed on one A100 GPU, elapsing 69 hours, equating to a little less than 2.9 days. In total, the entire evolution process elapsed 190 hours, 7.92 days, on one A100 GPU; a reduction by over 90% in time elapsed when compared to both NASNet and AmoebaNet.

The fitness results from the cascading mutation only genetic algorithm are depicted in Figure 5.11.

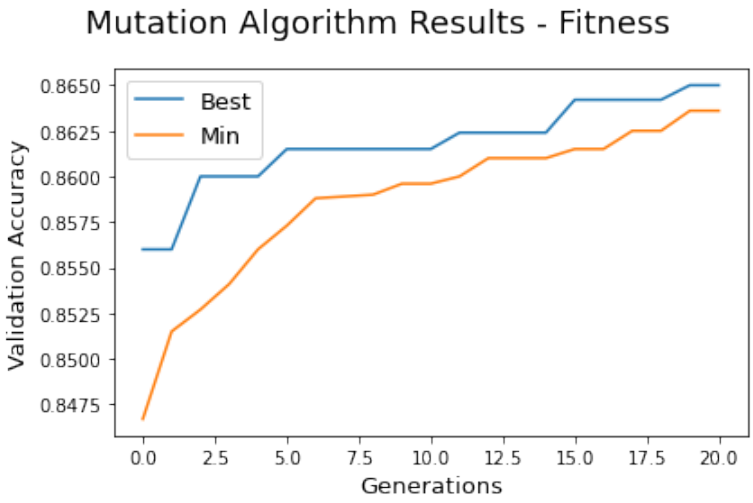



Figure 5.11: Cascading Mutation Genetic Algorithm Fitness Results

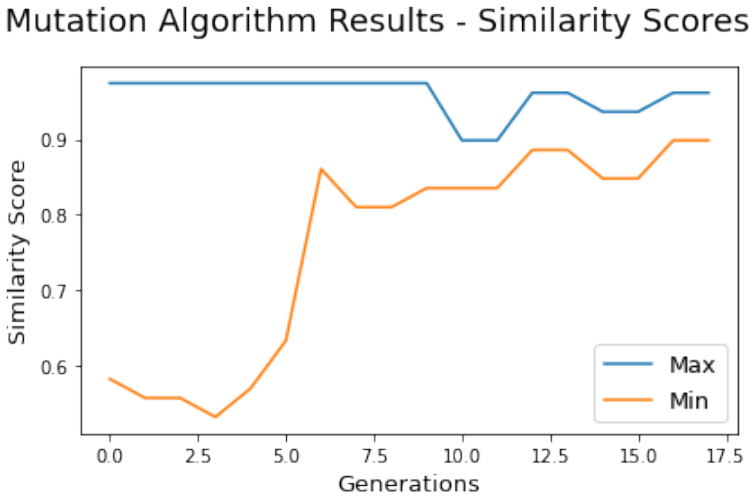



Figure 5.12: Cascading Mutation Genetic Algorithm Similarity Results

Figure 5.13 showcases the results from the mutation only algorithm from a different perspective. Figure 5.13 showcases a scatter plot of the cumulative searched models during the local search process along with their validation accuracy and number of parameters from their projection in the evaluation strategy.

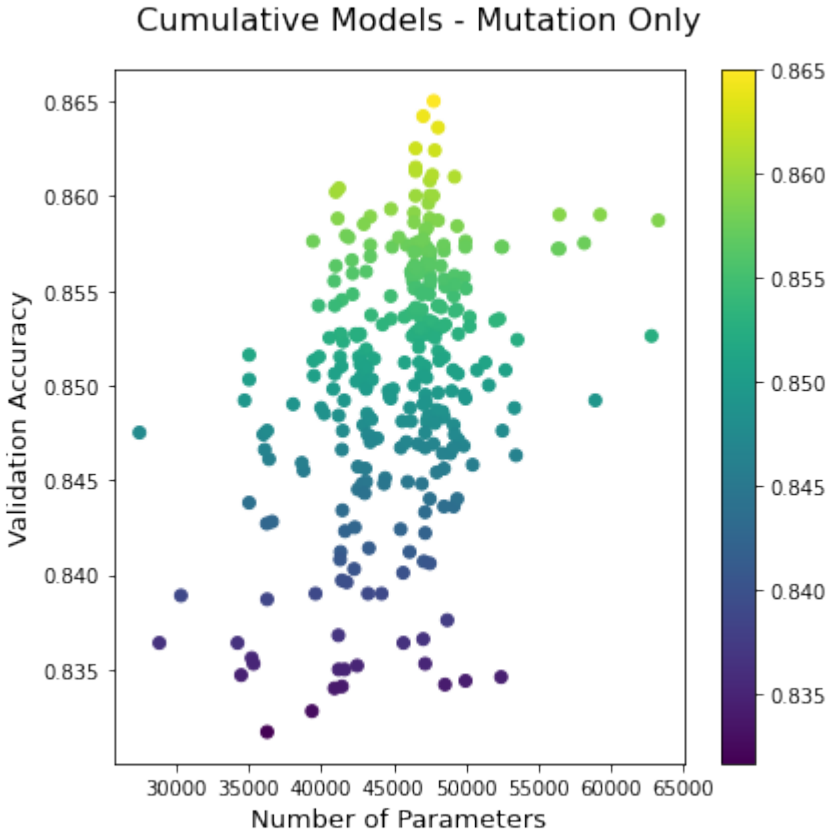



Figure 5.13: Cascading Mutation Genetic Algorithm Cumulative Model Results

## 5.3 Best Model Scaling

The best model was scaled to a various set of projections in order to obtain a full picture of its potential. The model was scaled to $n = 2$, $n = 4$, and $n = 6$ with various starting filters. Each model was trained using the entire 50,000 training images available from the CIFAR10 dataset and utilized the 10,000 test images as validation, as is common for comparing models on CIFAR10. Evolution did not utilize the test images as validation in order to ensure model transferability. Each model trained upon this dataset partition utilizing the CutMix image augmentation described in section 2.6.6 in order to encourage generalization, for a total of 900 epochs with a batch size of 512, equating to roughly 88,000 mini batch weight updates. In order to speed up convergence, AdamW optimizer was utilized along with one shot cycle learning rate with a minimum learning rate of 0.001 and a maximum learning rate of 0.05. Table 5.6 showcases all model projections in terms of number of filters and number of parameters in millions. Table 5.7 showcases all model projections in terms of dropout percentages and weight decay values.

Table 5.6: **Model Scaling - Number of Filters and Parameters**

| $n$ | Cell Module 1 | Cell Module 2 | Cell Module 3 | Parameters (Millions) |
|---|---|---|---|---|
| 2 | 4 | 8 | 16 | 0.0477 |
| 2 | 8 | 16 | 32 | 0.1845 |
| 2 | 16 | 32 | 64 | 0.7247 |
| 2 | 32 | 64 | 128 | 2.8728 |
| 2 | 64 | 128 | 256 | 11.4391 |
| 4 | 8 | 4 | 8 | 0.3314 |
| 4 | 16 | 32 | 64 | 1.3035 |
| 4 | 32 | 64 | 128 | 5.1702 |
| 6 | 16 | 32 | 64 | 1.8823 |
| 6 | 32 | 64 | 128 | 7.4675 |

Table 5.7: **Model Scaling - Dropout Percentages and Weight Decay**

| $n$ | Cell Module 1 | Cell Module 2 | Cell Module 3 | Last After | Weight Decay |
|---|---|---|---|---|---|
| 2 | 5% | 10% | 15% | 20% | $1e-7$ |
| 2 | 7.5% | 12.5% | 17.5% | 22.5% | $5e-7$ |
| 2 | 12.5% | 17.5% | 22.5% | 27.5% | $5e-7$ |
| 2 | 30% | 35% | 40% | 50% | $5e-6$ |
| 2 | 45% | 50% | 55% | 65% | $5e-5$ |
| 4 | 10% | 15% | 20% | 25% | $1e-7$ |
| 4 | 20% | 25% | 30% | 40% | $5e-7$ |
| 4 | 32.5% | 37.5% | 42.5% | 52.5% | $7.5e-6$ |
| 6 | 30% | 35% | 40% | 50% | $5e-6$ |
| 6 | 35% | 40% | 45% | 55% | $1e-5$ |

Each model was ran and evaluated three times, Figure 5.14 plots each model configuration with their respective number of cells per module, $n$, along with their number of starting filters (@$n$), validation accuracy, and number of parameters in millions.

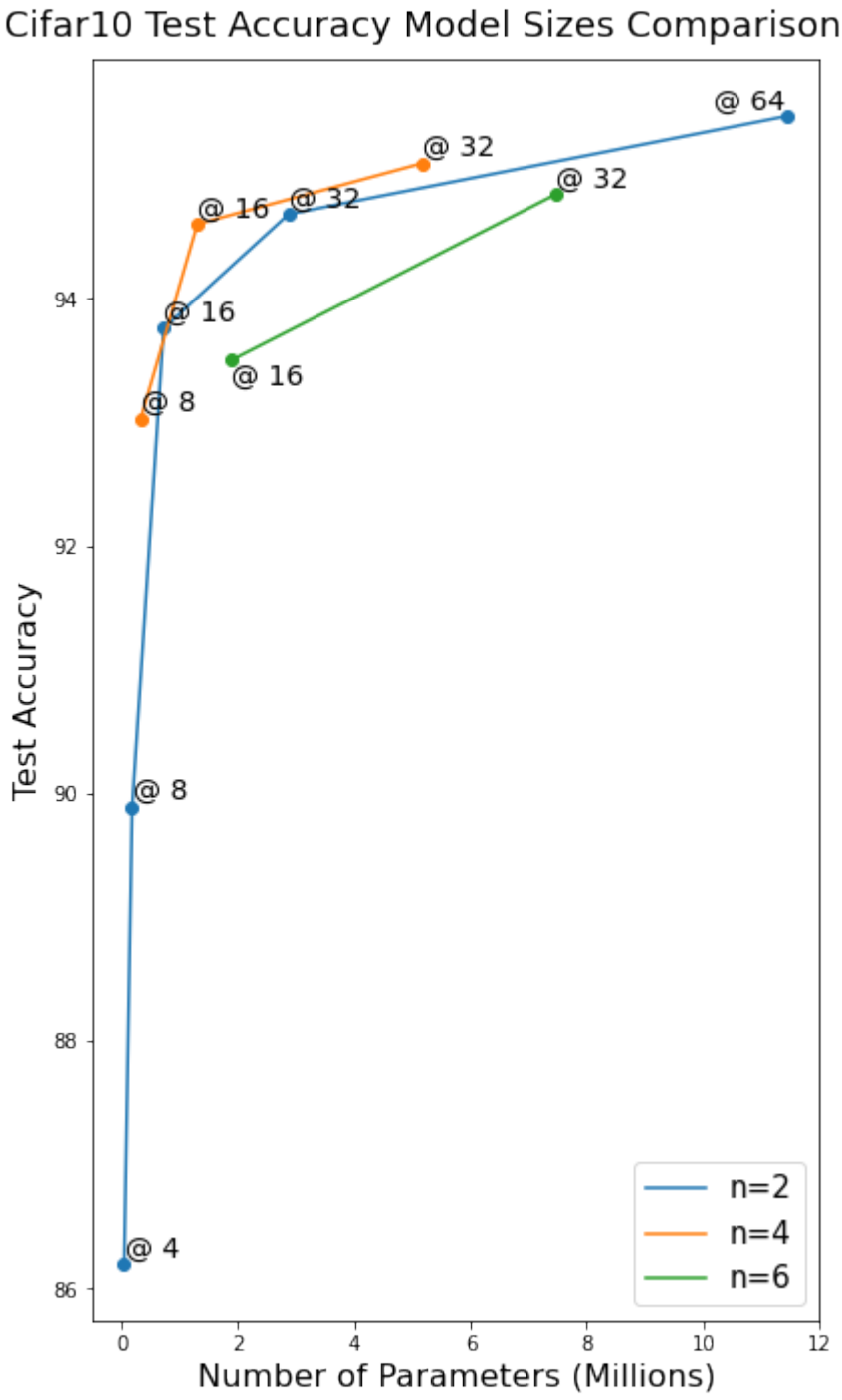



Figure 5.14: Model Projections - CIFAR10

Table 5.8 records the mean test accuracy for each of the model configurations as well as the results from other state of the art models for comparison.

Table 5.8: **CIFAR10 Model Comparison**

| **Architecture** | **Number of Parameters (Millions)** | **Test Accuracy** | **Search Cost (GPU Hours)** | **Search Method** |
|---|---|---|---|---|
| NASNetA ($n = 6$@768) [38] | 3.30 | 97.35 | 2,000 | Reinforcement |
| AmoebaNetA ($n = 6$@32) [22] | 2.60 | 96.60 | 2,000 | Evolution |
| AmoebaNetA ($n = 6$@36) [22] | 3.20 | 96.66 | 2,000 | Evolution |
| ResNet110 [12] | 1.7 | 94.54 | - | Human |
| ResNet1001 [12] | 10.2 | 95.38 | - | Human |
| WideResNet (w x 4) [36] | 8.70 | 95.03 | - | Human |
| DenseNet(k=24) [7] | 27.20 | 96.26 | - | Human |
| PyramidNet($\alpha$=48) [11] | 1.70 | 95.42 | - | Human |
| PyramidNet($\alpha$=84) [11] | 3.80 | 96.27 | - | Human |
| Ours $n = 2$@4 | 0.0477 | 86.20 | 190 | Evolution |
| Ours $n = 2$@8 | 0.1845 | 89.89 | 190 | Evolution |
| Ours $n = 2$@16 | 0.7247 | 93.76 | 190 | Evolution |
| Ours $n = 2$@32 | 2.8728 | 94.68 | 190 | Evolution |
| Ours $n = 2$@64 | 11.4391 | 95.47 | 190 | Evolution |
| Ours $n = 4$@8 | 0.3314 | 93.03 | 190 | Evolution |
| Ours $n = 4$@16 | 1.3035 | 94.60 | 190 | Evolution |
| Ours $n = 4$@32 | 5.1702 | 95.09 | 190 | Evolution |
| Ours $n = 6$@16 | 1.8823 | 93.50 | 190 | Evolution |
| Ours $n = 6$@32 | 7.4675 | 94.84 | 190 | Evolution |

## 5.4 Best Model Transfer

From the results in Table 5.8, the best model configurations, relative to number of parameters, was selected to be $n = 4$@16 and $n = 4$@32, as $n = 4$@16 achieved 94.60% mean test accuracy with only 1.3 million parameters, while $n = 4$@32 achieved 95.09% mean test accuracy with only 5.17 million parameters. These two configurations were then transferred and tested on the CIFAR100 dataset utilizing the same training methodologies, learning rate schedule, dropout rates, etc, as CIFAR10. However, a new configuration was added to see the middle ground between $n = 4$@16 and $n = 4$@32. This new model configuration was

$n = 4@24$, where the dropout rates were the mean between $n = 4@16$ and $n = 4@32$, and the weight decay value was set to $5e-6$. The results for the three configurations, along with state of the art models for comparison, are depicted in Table 5.9.

Table 5.9: **CIFAR100 Model Comparison**

| **Architecture** | **Number of Parameters (Millions)** | **Test Accuracy** | **Search Cost (GPU Hours)** | **Search Method** |
|---|---|---|---|---|
| ResNet110 [12] | 1.7 | 75.67 | - | Human |
| ResNet1001 [12] | 10.2 | 77.29 | - | Human |
| WideResNet (widthx4) [36] | 8.70 | 77.11 | - | Human |
| DenseNet(k=24) [7] | 27.20 | 80.75 | - | Human |
| PyramidNet($\alpha$=48) [11] | 1.70 | 76.88 | - | Human |
| PyramidNet($\alpha$=84) [11] | 3.80 | 79.34 | - | Human |
| Ours $n = 4@16$ | 1.3035 | 71.1 | 190 | Evolution |
| Ours $n = 4@24$ | 2.933 | 73.26 | 190 | Evolution |
| Ours $n = 4@32$ | 5.1702 | 76.53 | 190 | Evolution |

# Chapter 6

# Discussion

## 6.1 Initial Algorithm Comparison

The proposed genetic and particle swarm algorithms were all successful at optimizing model architectures on the reduced gray scale version of the CIFAR10 dataset. The different components utilized between algorithms were split in performance. Aging across the board yielded worse results compared to their non aging counterparts. This could be due to the maximum set age limit was too low, causing the loss of better solutions. However, this poses another hyper parameter necessary to tune. As a result, future work could entail eliminate aging as a consideration when using the smaller population sizes utilized in this work. Parent vs. global level elitism was split in performance with respect to aging. Parent level elitism performed worse than global for aging in the genetic algorithm while being slightly better for the non-aging component. The reasoning behind this could be perhaps due to the fact that both aging and parent level elitism act as a form of regularization against convergence; therefore, when combined, the results are worse than their counterparts as too much regularization was applied.

For the PSO algorithms, all under performed when compared to the genetic algorithms. The best and min results for the PSO algorithms were all lower than that of the genetic algorithm variants while being at the same hyper parameter settings. A couple of reasons could be supplied to explain this deficient. First, unlike the genetic algorithm, the PSO algorithms utilize low level genotype transformations to explore the domain space. As a result, perhaps the newly designed representation for the NASNet was insufficient to represent the search space. Second, the PSO algorithm itself is insufficient to efficiently explore the new representation. Third, the most likely position, the PSO algorithms exhibited extremely fast pre-convergence, which can be seen as the mean similarity scores for all PSO algorithms were between 85% and 90%, almost 20% higher than the genetic algorithm counterparts. Future work needs to be performed in order to solidify this claim. From the introduction of the Adam optimizer, the concept of scalable learning rates could be intermingled with the PSO algorithms in order to prevent pre-convergence by scaling the $w$, $c_1$ and $c_2$ coefficients based upon the similarity scores. From the variants of the PSO algorithms, gbest PSO outperformed subswarm PSO in all aspects. Despite breaking down the search space into two different sections to independently optimize, subswarm PSO struggled to optimize its particles. The reasoning is unknown for why this has occurred. Tt could be explained by the creation of the models for evaluation. Because each subswarm only deals with a particular section of the genome, each particle is evaluated by combining the best particle from the other subswarm to create the entire genome. However, the best particle from the other subswarm was defined to be which particle yielded the best fitness score, where that particles fitness score was created by combining that particle with another particle from the other subswarm. As a result, it cannot be determined whether the yielded

fitness score was mainly contributed by either the current particle or its counterpart subswarm particle. As a result, future work could entail using either random or proportional selection when choosing the particle from the other subswarm in order to reduce this concern.

## 6.2 Evolution of Chosen Algorithms

The full-scale global evolution phase was extremely successful, evaluating 1,000 models in 121 hours. The secondary phase of local search utilizing the cascading mutation only genetic algorithm was successful, but not only slightly when compared to the full-scale global phase, as the best fitness only increased by 1% after 69 hours of evolution. Future work could entail eliminating this secondary phase and evolving the global evolution phase further, as the time elapsed for the mutation only algorithm could have been utilized to evaluate another 570 models during the global search phase.

### 6.2.1 Global Search Phase

The best algorithm was then ran for full scale evolution for the global search phase. The results are depicted in Figures 5.8, 5.9, and 5.10. From Figure 5.8, the genetic algorithm for the global level search was successful at evolving the newly designed NASNet search space on the CIFAR10 dataset by increasing the best fitness value from less than 80% to 83%. In addition, the min fitness value finished past 82%. From Figure 5.9, the similarity scores were given, showcasing a moderate level of convergence during the global search phase. It is interesting to note that the min fitness was a little over 1% less than the final best fitness, which could indicate population convergence; however, the final mean similarity

score was not extremely large, less than 70, indicating that many of the solutions present in the final generation are similar in terms of most building blocks but still are very distinct. Figure 5.10 plots the max and mean ages from the global search phase, showcasing that the algorithm held onto one individual for almost half the lifespan of the algorithm. In addition, it appears that it went through a few trends of holding onto very old individuals before replacing them.

### 6.2.2 Local Search Phase

After the global search phase, the cascading mutation only genetic algorithm was utilized for the local search phase. The results are depicted in Figures 5.11, 5.12, and 5.13. From Figure 5.11, the mutation only genetic algorithm was successful at evolving the final generation by pushing it from a final best validation accuracy of 85.72% to 86.50%. The algorithm showcased good convergence, as to be expected by the cascading nature, as the final min fitness almost approaches the final best. Another support to this claim is Figure 5.12, which reveals that the min similarity score starts to approach 90% from 60%. Figure 5.13 plots the cumulative models found during the local search phase. As one can see, there does not seem to be any obvious skew in favoritism between small and large models, in terms of number of parameters, which is beneficial as it showcases that the search space is well built to not bias in any one direction. The best model trained achieved 86.5% validation accuracy while being around the mean for number of parameters.

## 6.3 Best Model

The normal cell for the best model architecture can be seen in Figure 6.1, which showcases each nodes connection along with their operation and the contents of

the After to the right hand side. From Figure 6.2, one can see that out of the six operations available, three of them are 1x5 5x1 convolutional kernels, showcasing that the evolution heavily favored this operation. By assessing the Afters, one can see that all Afters contain both batch normalization and dropout, which were allowed to turned on or off, indicating that the algorithm heavily favored these operations in the normal cell. The ordering is split between activation then batch normalization and batch normalization then activation, while being uniform in terms of activations found. As a result, not much can be concluded from those two aspects. Three of the six possible connections utilize the previous connections input, therefore it seems that the normal cell relies more upon the direct connection from the previous cell.

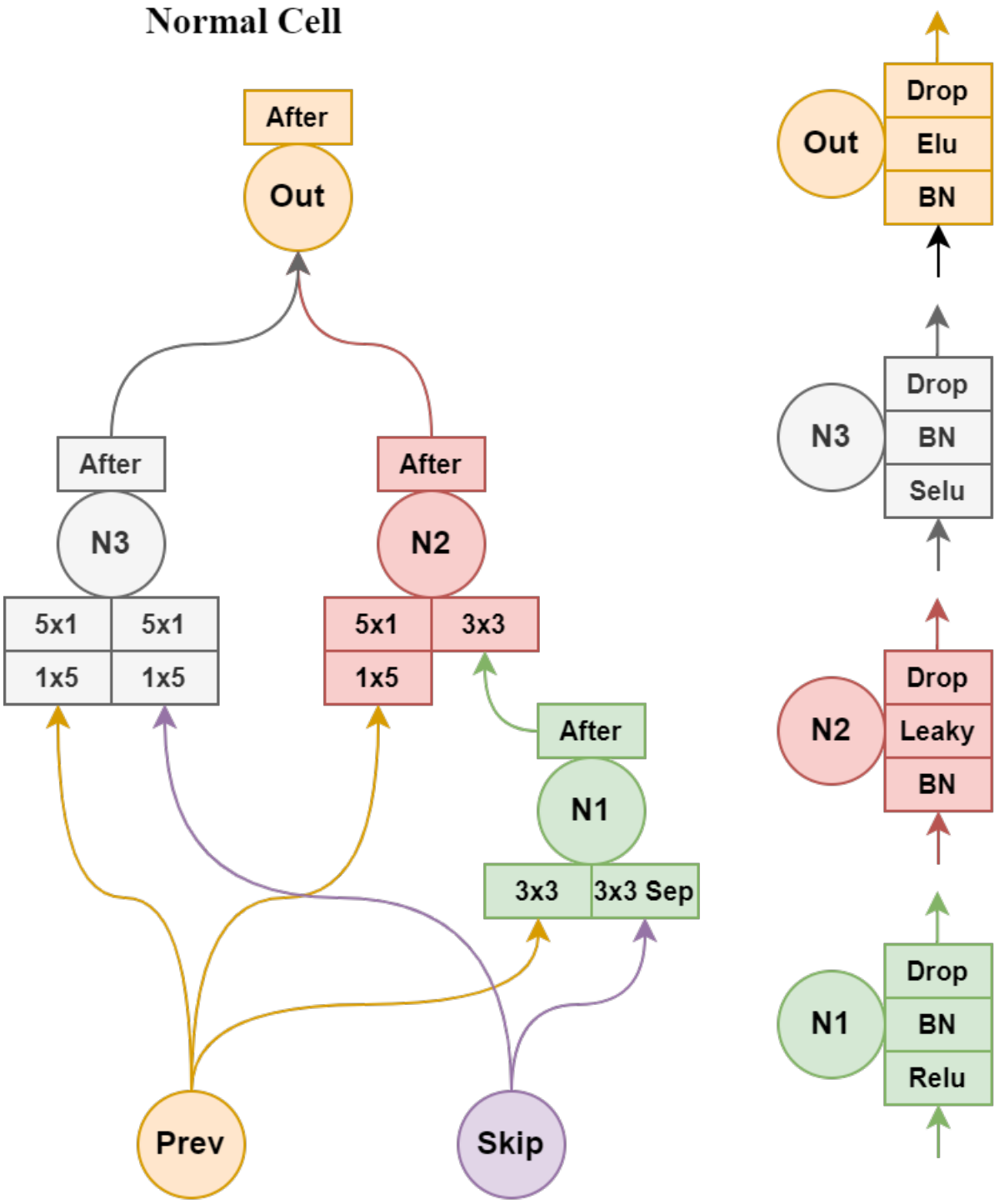



Figure 6.1: Best Model - Normal Cell

The reduction cell for the best model can be seen in Figure 6.2, which showcases each nodes connection along with their operation and the contents of the After to the right hand side. Unlike the normal cell, the reduction slightly favored the average pooling layer, while also slightly favoring convolution layers with larger kernels such as the 5x5 and again 1x5 5x1. By assessing the Afters, one can see that all Afters contain batch normalization, which were allowed to turned on or off, indicating that the algorithm heavily favored these operations in the reduction cell. The inclusion of dropout is split between two included and two not, therefore it cannot be concluded whether the algorithm favored, or not favored, the dropout layer. However, three of the four Afters utilize a batch normalization then activation ordering, showcasing that the algorithm favored this structure for the reduction cell. Three of the six possible connections for the cell utilize the output from hidden node state one, revealing that the reduction cell heavily relies upon the calculation from this node.

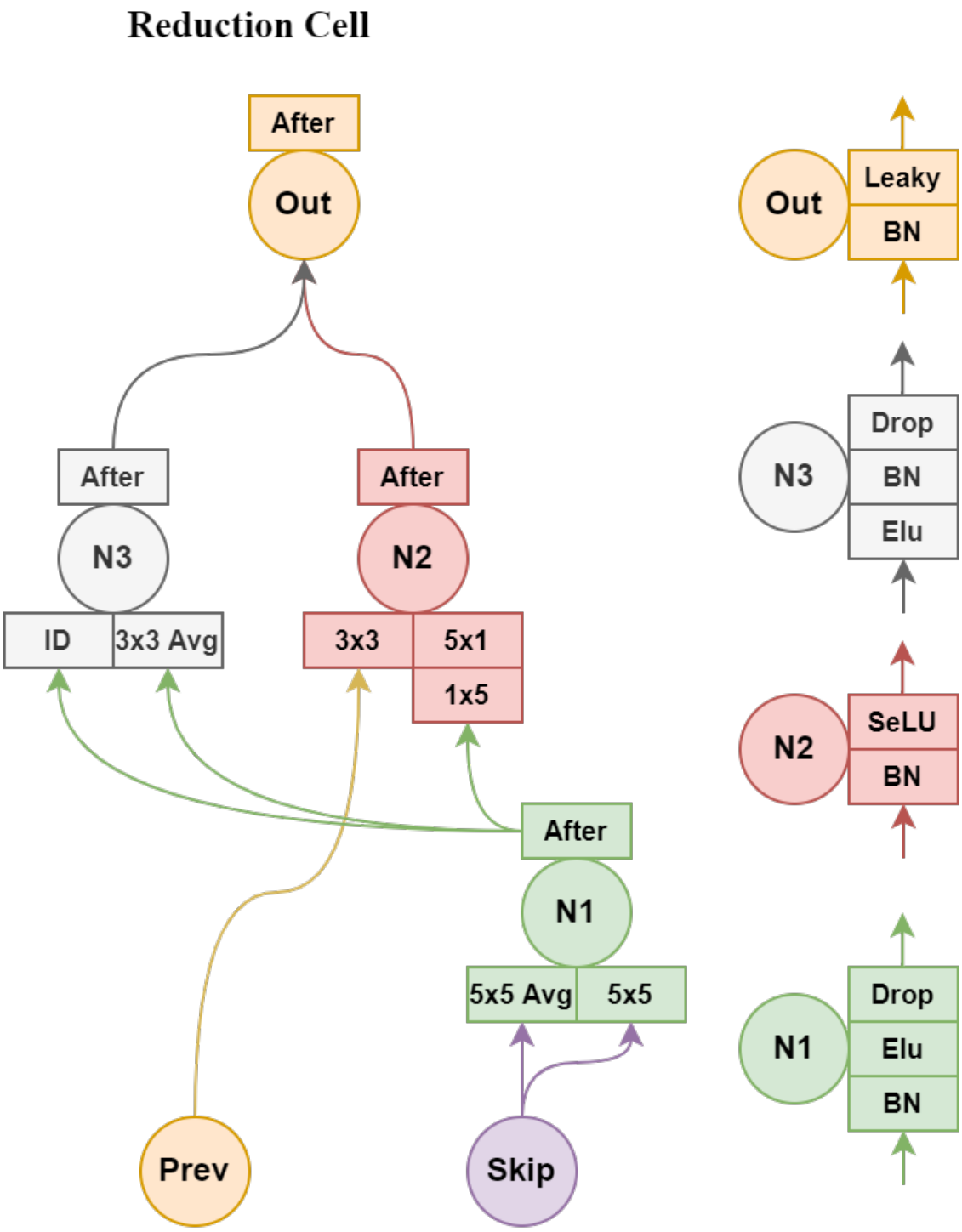



Figure 6.2: Best Model - Reduction Cell

The final model after evolution heavily relied upon factored 1x5 5x1 convolutions in the normal cell. This could be a reason to extend the NASNet search space to include more factorized operations such as 1x7 7x1 or stacked 1x5 5x5 5x1 operations, or perhaps include a 1x$n$ $n$x1 layer, where $n$ is also optimized in the node.

## 6.4 Best Model Scaling

Projecting the final model for comparison on the CIFAR10 dataset yielded very similar results to other state of the art models such as ResNet, WideResNet, and smaller versions of PyramidNet; however fell short when compared to AmoebaNet and NASNet, as one can see from Table 5.8. The reasoning behind this occurrence could be explained by three observations. First, the proposed evolution process evaluated less than 15% of the total number of models evaluated in AmoebaNet and NASNet; therefore, it can be expected that the final model does not match the refined searches of the other NAS systems. Second, only three hidden state nodes were utilized instead of five as in AmoebaNet and NASNet. The original reasoning behind this reduction in hidden state nodes was to speed up evaluation as training with less hidden state nodes; however, perhaps three hidden nodes is insufficient to create extremely good models. As a result, future work could entail evolving using five nodes; or, evolve using three nodes but then project the final population to five nodes and then perform a secondary level of evolution. In this way, the evolutionary phase can be split into two sections, one performed using three nodes, which reduces the complexity of the optimization problem at hand while also reducing the time required to train models, and the second projected the final to five nodes for a secondary phase of evolution in order to obtain a

better model. Lastly, the inclusion of dropout posed as a problem due to its requirement of a dropout percentage. As a result, model results can vary based upon the setting of these dropout percentages. Future work could entail either eliminating dropout layers from within the cells or establish a set function that yields the dropout percentage based upon the model at hand, instead of manually tuning the parameters.

## 6.5 Best Model Transfer

Transferring the model to CIFAR100 yielded slightly worse results than state of the art models, as one can see from Table 5.9. It is unclear whether this was observed due to the hyper parameter settings for the dropout layers, which can be seen as a secondary optimization problem, or that the evolution algorithm heavily over fitted the model architecture to the CIFAR10 dataset. Further work would need to be performed in order to solidify either claim. Future work could entail evaluating a model on multiple datasets, averaging their results, in order to ensure model transferability; however, training a model multiple times greatly increasing the computational costs.

## 6.6 Future Work

Lastly, in the context on future work, the NASNet search space could be further expanded to include components from other state of the art models such as PyramidNet [11], which slowly increasing the number of channels per cell instead of sharply increasing with each reduction cell, or DenseNet [7], which utilizes skip connections from all previous cells, or ShuffleNet [37], which utilizes grouped

convolutions. Although the NASNet search space was revolutionary at the time, better state of the art models have been created, whose research can be combined with the NASNet search space to create a newer search space with the ability to yield even more efficient and powerful models.

# Chapter 7

# Conclusion

The emergence of the NASNet search space was revolutionary as it allowed for NAS systems to eliminate the need of concern to allow for model scalability during the searching process, as each model could be scaled independently from the searching process. It has provided the cornerstone from which this work derived its search space. However, the NASNet search space lacked the ability to express optimization of non-convolutional operational layers, such as batch normalization, activation, and dropout. This work introduced these components into the NASNet search space to allow for the search algorithm to select the best possible combination of these non-convolutional operational layers on a per node basis. In addition to changing the NASNet search space, it was projected into a fixed length continuous floating point dimensional array. Previously defined NAS systems created models at the phenotype level, using a recurrent neural network controller that recursively produced networks through its softmax layer, or using a directed graph as in AmoebaNet. Standard non-classical optimization algorithms, such as PSO, leap frog, hill climber, low level genetic algorithms, and others, all require a fixed length continuous floating point dimensional array for

optimization. As a result, this limitation on the current NASNet search space representation narrows down the possible optimization algorithms that can able be applied. In this work, a fixed length continuous floating point dimensional array of the altered NASNet search space was proposed. The details of its creation are found in section 4.2.

After the creation of the new NASNet search space, two non-classical optimization algorithms were selected for optimization, a genetic and a particle swarm algorithm. The genetic and particle swarm algorithms encompassed many different variations. Variations used to control convergence, such as parent level vs. global elitism; variations for controlling regularization through aging; variations used to control exploration through population size; and, variations at the algorithmic level with different major components, such as cascading genetic algorithm and gbest subswarm optimization. Unfortunately, all possible combinations were not able to be tested full scale due to the computational requirements. However, all were tested on a preliminary gray scale reduced version of CIFAR10. From those results, the best algorithm was selected to be a genetic algorithm utilizing global level elitism, without aging, and a population size of 16.

With the goal in mind of reducing the computational cost of searching the new NASNet search space, as previous implementations evaluated 20,000 models, requiring 2,000 GPU hours, the search phase was limited to only 1,300 fitness function evaluations. From which, two independent search phases were constructed, a global and a local search. The global search phase had the goal in mind of exploration the search space while exhibiting slight convergence in the final population. The goal of the local search phase was to narrow down the final population from the global search in order to refine around the best solution. The best genetic

algorithm from the preliminary gray scale reduced version of CIFAR10 was selected to be search algorithm for the global search phase. Because the goal of local search phase was convergence, a cascading mutation only was introduced and selected to be the search algorithm of the local search phase. The results from the global search phase showcase that the newly designed NASNet search space, the new representation, and the designed genetic algorithm, were success at representation and evolution as the best validation accuracy increased from a starting position of 79.7% to 83% over the course of only 121 GPU hours on one A100 GPU. In addition, the results from the local search phase showcase that the cascading mutation only genetic algorithm was able to refine the best model from 85.65% to 86.50% over the course of 69 GPU hours, while also increasing min similarity score from 60% to 90%.

The best model was then projected to a number of different model configurations, varying the number of cells per normal module and the number of starting filters. The best model configurations obtained from the CIFAR10 dataset was considered to be $n = 4@16$ and $n = 4@32$, as $n = 4@16$ achieved 94.60% mean test accuracy with only 1.3 million parameters, while $n = 4@32$ achieved 95.09% mean test accuracy with only 5.17 million parameters. The results were very comparable to state-of-the-art models, as the configurations were better than ResNet110 and WideResNet, but fell short compared to NASNet-A and AmoebaNet-A. The best configurations were then transferred to the CIFAR100 dataset to assess model transferability. The results indicate that the model configurations fell slightly behind state-of-the-art models, indicating that the model either lacked the ability to efficiently transfer datasets or the hyper parameters for regularization were not sufficient. This question was left for future work.

In conclusion, the newly proposed representation for the proposed altered

NASNet search space were designed well enough for the chosen search algorithms to search and explore the domain space. It is hoped that this new representation can encourage other non-classical optimization algorithms to tackle the NASNet search space. In addition, the methodologies incorporated for speeding up model evaluation, along with the selected genetic algorithms, were able to yield state of the art models while requiring less than 10% of computation required by AmoebaNet and NASNet.

# Bibliography


[1] Franciois Chollet. "Xception: Deep Learning with Depthwise Separable Convolutions". In: *CoRR* (2017).

[2] Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. "mixup: Beyond Emperical Risk Minimization". In: (2018).

[3] Deng et al. *Imagenet: A large-scale hierarchical image database*. 2009, pp. 248–255.

[4] Terrance DeVries and Graham W. Taylor. "Improved Regularization of Convolutional Neural Networks with Cutout". In: (2017).

[5] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* (2011).

[6] Andries P. Engelbrecht. *Computational Intelligence, An Introduction*. John Wily and Sons, Ltd, 2007.

[7] Z. Liu G. Huang and K. Q. Weinberger. "Densely Connected Convolutional Networks". In: *CoRR* (2016).

[8] Divya Gaur, Joachim Folz, and Andreas Dengel. "Training Deep Neural Networks Without Batch Normalization". In: *CoRR* (2020).

[9] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. “Deep Sparse Rectifier Neural Networks”. In: *International Conference on Artificial Intelligence and Statistics* 15 (2011).

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[11] Dongyoon Han, Jiwhan Kim, and Junmo Kim. “Deep Pyramidal Residual Networks”. In: *CoRR* (2017).

[12] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *CoRR* (2015).

[13] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep network Training by Reducing Internal Covariate Shift”. In: (2015).

[14] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *ICLR* (2017).

[15] Gunter Klambauer, Thomas Unterthiner, and Andreas Mayr. “Self-Normalization Neural Networks”. In: *Neural Information Processing Systems* (2017).

[16] Alex Krizhevsky. “Learning multiple layers of features from tiny images”. In: (2009).

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *NIPS* (2012).

[18] Anders Krogh and John Hertz. “A simple Weight Decay Can Improve Generalization”. In: (1991).

[19] George Kyrakides and Konstantinos Margaritis. “An Introduction to Neural Architecture Search for Convolutional Networks”. In: *CoRR* (2020).

[20] Ilya Loshchilov and Frank Hutter. “Decoupled Weight Decay Regularization”. In: *ICLR* (2019).

[21] YU. E. Nesterov. “A Method of Solving A Convex Programming Problem with Convergence Rate O($1/k^2$)”. In: (1983).

[22] Esteban Real et al. “Regularized Evolution for Image Classifier Architecture Search”. In: *CoRR* (2018).

[23] Shibani Santurkar et al. “How Does Batch Normalization Help Optimization?” In: *CoRR* (2019).

[24] J. Schulman et al. “Proximal Policy Optimization Algorithms”. In: (2017).

[25] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks For Large-Scale Image Recognition”. In: *ICLR* (2015).

[26] Leslie Smith and Nicholay Topin. “Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates”. In: *CoRR* (2018).

[27] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* (2014).

[28] Kenneth O. Stanley and Risto Miikkulainen. “Evolving Neural Networks through Augmenting Topologies”. In: *Evolutionary Computation* 10.2 (2002), pp. 97–127.

[29] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. “Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning”. In: *CoRR* (2016).

[30] Christian Szegedy et al. “Going Deeper with Convolutions”. In: *CoRR* (2014).

[31] Christian Szegedy et al. “Rethinking The Inception Architecture for Computer Vision”. In: *CoRR* (2015).

[32] Djork-Arne Clevert andThomas Unterthiner and Seep Hochreiter. “Fast and Accurate Deep Netowrk Learning by Exponential Linear Units (ELUS)”. In: *International Conference on Artificial Intelligence and Statistics* (2016).

[33] Zhengjiya Wu and Jianzhong Zhou. “A Self-Adaptive Particle Swarm Optimization Algorithm with Individual Coefficients Adjustments”. In: *IEEE* (2007).

[34] Bing Xu, Naiyan Wantand Tianqi Chen, and Mu Li. “Empirical Evaluation of Rectified Activations in Convolution Network”. In: (2015).

[35] Sangdoo Yun et al. “CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features”. In: (2019).

[36] S. Zagoruyko and N. Komodakis. “Wide Residual Nnetwork”. In: *BMVC* (2016).

[37] Xiangyu Zhang et al. “ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices”. In: *IEEE* (2018).

[38] Barret Zoph et al. “Learning Transferable Architectures for Scalable Image Recognition”. In: *CoRR* (2018).