VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Genetic Programming for Evolving Neural Networks for Image Classification

Andrew McGhie

Supervisors: Bing Xue, Mengjie Zhang

Submitted in partial fulfilment of the requirements for Bachelor of Engineering with Honours in Software Engineering.

Abstract

In this report, a new approach to automatically searching for architectures of Convolutional Neural Networks (CNNs) is outlined (GPCNN). This approach utilizes Genetic Programming (GP) which uses trees to represent architectures of CNNs so that crossover operations can be applied to the individuals. Three modifications were also developed. The first was Partial Subtree Crossover which is a new crossover operation that exchanges partial subtrees. The second is Variable Fitness Evaluation where some individuals were trained for more epochs than others. The last was Building Block initialization where the initial population was built with higher-level building blocks. The algorithm was tested on the CIFAR-10 that contains 60,000 instances spread across 10 classes. GPCNN achieved 81.2%. GPCNN with Partial Subtree Crossover achieved 82.2%. GPCNN with Variable Fitness Evaluation achieved 80.5% and GPCNN with Building Block Initialization achieved 78.1%.

Acknowledgments

I would like to thank my supervisors Bing Xue and Mengjie Zhang who both provided me with amazing support and guidance allowing me to complete the project to a high standard. I would also like to thank Ying Bi and Sharon Gao for their help with the project.

Contents

Figures

Chapter 1

Introduction

Analysis of images is an area of machine learning with many real-world applications. Examples include: fruit quality assessment, number plate identification, facial identification, and medical imaging analysis. Image analysis can be achieved more rapidly using computers with machine learning software than by human analysis and the accuracy levels achieved are often higher.

1.1 Motivations

Convolutional Neural Networks (CNNs) are complex machine learning algorithms that are becoming larger and more difficult to build and understand. Increasing complexity makes it challenging for humans to design improved architectures and a high level of domain knowledge is required. Automation of the architectural design of CNNs using Neural Architecture Search (NAS) algorithms will potentially overcome these limitations. NAS is an approach that automatically generates the architecture of a neural network.

There are two main ways that NAS can generate the architectural design of a CNN, either by the use of an Evolutionary Computation (EC) algorithm (NeuroEvolution) or by Reinforcement Learning. [1] evaluated five of each type of algorithm and found that EC based algorithms trained faster and were more accurate on average compared with Reinforcement Learning based algorithms. Computation time of the final models was also compared and it was found that models developed by EC had a smaller computational cost than models developed with a Reinforcement Learning or Random search approach. NAS has been used with increasing success in recent years due to the increase in computational power available making their use more feasible.

Automatically designing the architecture of a CNN has significant advantages in requiring neither domain knowledge of the problem or an in-depth knowledge about CNNs. It also has the potential to find previously unknown knowledge regarding applying CNNs to the domain.

Genetic Programming (GP) is an EC algorithm which has the advantage of representing individuals in the population as trees. This type of representation has more complex relationships between the parts of each individual and is important as state-of-the-art CNNs are no longer a sequence of layers, but rather Directed Acyclic graphs (DAG). Having more complex relationships in individuals makes it easier to create DAGs from the complex relationships in the individuals. CNNs use architectures with complex connectivity to allow the training of deeper networks. An example is the skip connections used by ResNet [2]. These are identity connections that are in parallel with the sequential layers allowing for the gradient used in backpropagation to not be influenced by weights in the following layers.

Trees are closer to DAGs than other encoding strategies that such as binary strings and sequences of layers. A disadvantage of encoding strategies that use DAGs directly is that they are unable to use crossover operators due to the lack of a crossover operator that can handle sections of layers with multiple inputs and outputs. Therefore, tree based GP has the unique position where complex connectivity can be expressed and it is possible to apply crossover operators. Applying GP to image classification is complex as images usually have a large amount of data to process with the individual pixel value combined with the large search space from using trees causes GP on image data to be computationally expensive.

1.2 Goals

The aim of this project is to generate an encoding strategy that can be used where the architecture of the neural network is evolved with a crossover operation that can capture high performing groups of layers. To achieve this, GP was used to evolve CNN architectures to solve an image classification problem. Here is a number of goals this project attempts to achieve:

- Use GP where individuals can represent complex architectures and are compatible with crossover. The idea of this is to allow the ability to spread valuable features from high fitness individuals to other individuals. This idea comes from the Building Block Hypothesis or Schema Theorem of Genetic Algorithms (GA) [3].

- To develop a crossover operator based on partial trees that will allow the GP algorithm to efficiently explore the space around an individual.

- To develop a new method of evaluating individuals that would allow the GP algorithm to accurately explore the search space while not greatly extending the time required to evaluate the fitness of an individual.

- To initialize the population with a supply of building blocks that perform well. This comes from the idea of the Building Block Hypothesis that GP searches hierarchically by combining building blocks [3]. There are a number of building blocks that have been shown to perform well from handcrafted architectures.

1.3 Major Contributions

- This project proposes the GPCNN algorithm that uses GP to evolve complex CNN architectures from a range of different types of layers and connections. A new and powerful encoding strategy is used to convert trees into CNNs that use complex connectivity rather than sequential layers. The results show that it is a viable approach, although more work will need to be done to achieve accuracy similar to current state-of-the-art approaches.

- This project also proposes a novel crossover technique that can exchange random partial subtrees rather than whole subtrees. The main advantage of this new crossover technique is that to change layers at the end of the network can require changing all layers that come before that layer. Exchanging partial subtrees makes it easier to search the space close to the fittest individuals. This had a minor effect on the accuracy of GPCNN even though the crossover was significantly more destructive.

- The training of the CNNs produced by individuals is the main factor contributing to the runtime of the algorithm as it is required to calculate the fitness values of the individuals. There is a trade-off between the accuracy of the fitness values and the time taken to calculate the fitness values. A less time consuming approach would be to train some individuals for longer to get an accurate measure of fitness and to test how susceptible they are to overtraining. This led to the GPCNN algorithm converging much faster while achieving similar accuracy to the base GPCNN.
- A different way to build the initial population is also proposed which randomly combines high level blocks of layers that have been found to work well together from hand-made networks. These blocks are converted into partial subtrees made up of functions from the function set used during evolution. This approach differs from traditional methods of creating the initial population for GP where the trees are built by choosing nodes from the function and terminal set that is used during the evolution. Building block initialization achieved lower accuracy than the base GPCNN, possibly due to a lack of diversity in the population during the evolutionary process.

1.4 Report Organisation

Chapter 2 provides an overview of the different techniques that were used in the project. Chapter 3 outlines a GPCNN that is a novel way to evolve CNN architecture. Chapter 4 describes a new partial subtree crossover operator. Chapter 5 outlines Variable Fitness Evaluation, a different approach to training individuals. Chapter 6 discusses building block initialization.

Chapter 2

Background

2.1 Image Classification

Image classification falls under a broader category of computer vision that aims to allow computers to interpret natural-scene images based on the content of an image [4]. Examples included labeling images of cats and dogs or images of digits. Enabling computers to know what is in an image by image classification is an important part of computer vision as it allows further actions to be undertaken based on that information. Other aspects of computer vision include feature extraction, feature construction and object detection for finding the position of objects.

There are a number of issues involved in image classification. The high dimensionality in images leads to the curse of dimensionality [5], which is where the signal-to-noise ratio is low and inhibits training of models. There is often a large number of classes and a wide range of images that belong to each class, creating a very complicated problem to solve. A further important issue is the ability to detect objects in images regardless of their position and orientation.

2.2 Convolutional Neural Networks

CNNs are a form of neural networks that were inspired by the connectivity of neurons in the visual cortex of cats [6]. They are a higher order neural network that has layers that use cross correlations between different filters and images in the dataset to calculate feature maps.

Currently, the state-of-the-art methods for image classification are all CNNs [7, 8, 9] which make use of multiple convolutional layers with multiple convolutional filters. The famous CNN architectures are all hand-crafted. An example is the ResNet architecture [2] which implemented skip connections to allow residual learning to avoid the vanishing gradient problem. This idea forms the basis of many other architectures such as the ones used in [10, 11, 12]. Additionally deep CNNs have become more viable as computational power increases and the necessary techniques to train them are being developed. These techniques include regularization techniques such as dropout [13] and batch normalization [14] and techniques to avoid vanishing gradient such as Relu activation [15] and various different architectures many of which are discussed throughout this report.

In recent years the state-of-the-art CNNs have been growing in complexity and moving away from a linear architecture [16, 17] to more complex graph structured architectures [18, 19, 20, 21, 22]. This greatly increases the search space of CNNs making them more expensive and harder to design by hand so they will be more likely to have suboptimal performance. Autonomous searching makes designing an architecture for a domain requires less knowledge as it is much easier for a human. The majority of the design process will be performed by a computer and thus requires minimal input.

2.3 Evolutionary Computation

EC is a field of artificial intelligence that is inspired by biological evolution [3]. EC algorithms consist of a population of individuals where each individual represents a solution to the problem. The individuals are changed overtime in such a way that the individuals in the population represent a better solution to the problem. This is a global search approach as opposed to the training of a neural network that tries to minimise the loss of the network. How well an individual solves a problem is defined by the user defined fitness function. Since the fitness function does not have to be differentiable, like the loss of a neural network due to the use of gradient descent, it is more flexible [23].

2.3.1 Genetic Programming

GP [24] is a form of evolutionary algorithm that represents individuals in its population as trees. For each generation run, it applies a selection operator to the population in order to select high performing individuals before creating new individuals by mating (through a crossover operation) or mutation. The selection operator uses the fitness value associated with each individual to select high performing individuals and remove low performing individuals. This value represents how well the individuals solve the given problem.

Individuals are typically represented as trees. The nodes in the individuals represent functions and the leaf nodes represent constant values called terminals. A tree as seen in Figure 2.1 would be evaluated as $max(x * x, x + (3 * y))$.

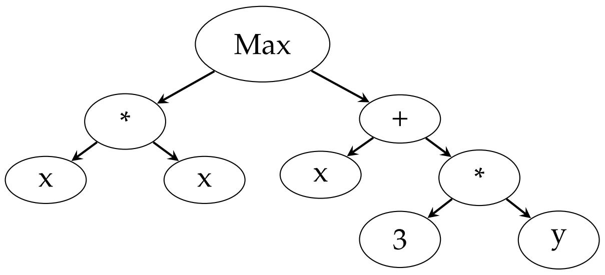


Figure 2.1: Example of a tree in GP

Selection

Tournament selection [24] is a common selection method used with GP. The fittest individual is chosen from a group of n randomly chosen individuals. It is further repeated until the required number of individuals has been selected. Elitism [24] is an operator that is used in conjunction with another selection method such as tournament selection. A number of the best individuals from one population are copied to the next generation to ensure that the genetic material is passed on.

Ordering of the fitness value compared with each other matters, however, the magnitude does not. GP algorithms can use estimations of the fitness values and achieve similar results to a GP algorithm using true fitness values. Using estimations is helpful when the true fitness value takes a long time to calculate. This can create a trade off between how fast a GP algorithm can evaluate an individual and how accurate the fitness value is. Quicker evaluation allows for the GP algorithm to search a greater portion of the search space. However, if the fitness values are incorrect, it will explore parts of the search space that do not perform well and become misguided.

There are a number of ways that estimations of the fitness values can be calculated when using neural networks as the individuals. The most common way is to train the individuals for fewer epochs than is needed for the networks to converge. Hype networks can also be used, being networks that predict the weights of another neural network rather than training that network through gradient descent. Proxyless NAS [25] had success with this method and managed to achieve a smaller search time of the algorithm than competing algorithms.

Crossover

Koza [24] outlined the basic crossover operator where two individuals are chosen at random to be the parents. A random node in each parent is chosen then the subtrees from those nodes are swapped between the parents to create two new individuals.

The usefulness of crossover in GP algorithms has been under some debate. However, [26] found that there was a difference between using a high crossover rate and a high mutation rate. Where high mutation rate tended to perform better with a smaller population size, crossover performed better with a larger population size.

Mutation

The commonly used mutation operator is the subtree mutation operator that works by randomly selecting one parent then selecting one node and replacing the entire subtree that has the node as the root with a newly generated subtree to create a new individual.

The process of applying the selection mutation and crossover operators are shown in Figure 2.2. Each generation consists of one iteration around the loop of evaluation, selection, mutation and crossover.

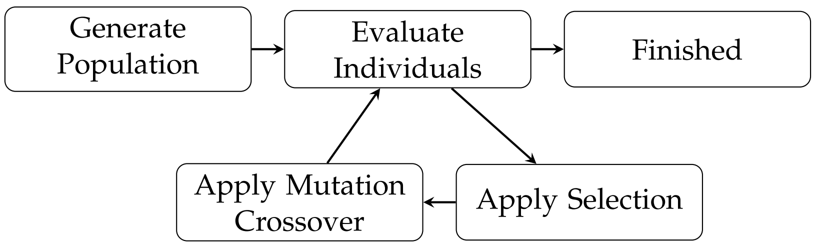


Figure 2.2: The GP process

Bloat

It is known that GP and other evolutionary algorithms that use variable length genotypes have a tendency of growing the length of the genotypes [27]. In the case where individuals represent neural networks, this bloat can greatly slow down the algorithm by causing the evaluation of each individual to take longer than is required.

One explanation of bloat by [27] is that bloat is caused by individuals protecting themselves from the destructive nature of the genetic operators by having a large portion of the individual that has no effect on the fitness of the individual. This is called neutral code. In the case of using CNN as individuals, this would mean that neutral code consists of layers that do not perform useful operations.

A way to avoid bloat is by using a double tournament [28], where the selection operator uses two tournaments. The first tournament on the fitness values finds two individuals and is then followed by a second tournament on these two individuals using the size of the individuals. The power of the size tournament is managed by the parsimony pressure [28] which is a penalty term that weakens the effect of the second tournament by randomly not choosing the winner of the tournament. Using a tournament on size puts pressure on the individuals in the population to have a high fitness and a low size.

Another method that bloat is controlled is by limiting the depth or size of the trees by limiting the size at initialization and reducing tree size when crossover or mutation makes them too large [24]. This way to control bloat is weaker as it relies on the limit to be configured to the size or depth of the globally optimal tree which is not known. If the limit is configured too high then bloat will exist, if too low then the final accuracy of the algorithm will be negatively affected. An advantage of limiting depth is it guarantees that all individuals will be under a certain size which can be useful to make sure requirements, such as memory, are not exceeded.

Search space

Tree-based GP usually has a much larger search space than other EC techniques such as GA [29] or Particle Swarm Optimization (PSO) [30] which use binary strings or sequences of numbers to represent individuals. The larger search space results from the more complex adjacency where each node can have a number of children as opposed to sequences of numbers where each item can have, at most, one item before and one afterwards.

Strongly typed Genetic Programming

An extension of GP is Strongly typed Genetic Programming [31]. This adds constraints to the children of nodes so that they have to match types defined by the node. Each child to a node has to match a given type which can allow constraints to be put on the structure of the individuals.

2.4 Related Work

2.4.1 Genetic Programming for image analysis

GP has been used to solve image classification problems in a number of ways.

Two Tier GP has been developed by [32] to classify images. The lowest tier used aggregation functions that calculate metrics over part of an image parameterised by terminals on the node. The higher tier used classification functions combining the values from the lower tier to create the predicted class value.

[33]used three-tier GP for image analysis. The bottom tier used a function set made of convolution and pooling functions and where the terminals were specific convolutional filters. The middle tier applied aggregation to the image to find one feature. The top tier applied threshold functions to classify the instances to a class. Convolutional operators were learnt as part of the evolutionary process of this three-tier GP.

There has been further work on this way of using GP by [34]. Multi-layer GP was proposed where region detection, feature extraction, feature construction and classification were performed by different layers on image data.

2.4.2 NAS

NAS is the name for a family of techniques that attempt to automate the design of artificial neural networks (ANN). NAS is an alternative to hand crafting the architecture. The main drawback of NAS is that a large computation time is required for evaluating neural network architectures, however, as available computational power has increased, so has our ability to use NAS for generating ANNs. Additionally, NAS requires less domain knowledge than hand-crafted networks. NeuroEvolution methods have been successfully applied to neural networks for the last two decades [35]. More recently hand-crafted networks have fallen behind NAS methods. The three best algorithms on CIFAR-10 [36] all use some form of architecture search to some degree. GPipe [7] achieved 99% with 557 million parameters and applied NAS on much larger networks than had previously been possible. EfficientNet [9] used a small grid search scaled them up baseline networks and achieved 98.9% accuracy with 64 million parameters. ProxylessNAS [37] used NAS with a hypernetwork to predict the weights on each individual to speed up searching and achieved 97.92% accuracy with just 5.7 million parameters.

The architecture of a neural network is considered to be a hyperparameter of a neural network model, therefore NAS falls under the broader category of hyperparameter optimization. Techniques used for hyperparameter optimization include random search [38] and Bayesian optimization [39]. However, these techniques either require a large number of evaluations or do not handle the complexity of the architecture.

2.4.3 NeuroEvolution on CNN

Utilizing NAS for generating deep CNN architectures has increased in popularity over recent years due to the increase in computational power available. It is computationally expensive to evaluate a deep CNN, making search strategies on deep CNN architecture even more expensive as a large number of architectures have to be evaluated to find an optimal result.

There are also a number of problems in representing neural networks with an encoding strategy for evolutionary algorithms. This is because modern deep neural networks are DAG rather than a linear sequence of layers. For example, GA uses a binary sequence that makes it difficult to represent a DAG while also being able to effectively apply mutation and crossover operators to the individual. A common way to work around this is to abstract complex groups of layers that have been found to perform as inseparable blocks of layers as seen in [40, 41, 42]. This reduces the search space of the algorithm by making the assumption that the blocks will be able to create an optimal solution for the problem.

2.4.4 Encoding Strategies

There are two families of encoding strategies: direct encoding and indirect encoding. Direct encoding is where the representation of the individuals are the layers and connection of the neural network. In the case of deep CNN, the individuals are DAGs and the operators in an EC algorithm would have to operate directly on DAGs. Indirect encoding uses a separate data structure to represent the individuals for the purpose of ensuring that the evolutionary operators are compatible with the individuals. In the case of GA [29] and PSO [30] based methods, binary strings or sequences are used. In GP [43] trees are typically used.

A large number of encoding strategies utilise binary strings to create complex neural network architecture. For example, [40] used Cartesian Genetic Programming where each layer was represented by the function id and the layers it was connected to. Another approach was used in [44] where the different layers had different numbers of parameters and the binary strings were joined together into a fixed length so that a CNN of varying length could be represented with a fixed length binary string. An approach that did not use binary strings and instead used a linked list approach is [42]. A linked list with the nodes representing blocks of layers had been found to be useful from handcrafted CNN architectures.

2.4.5 Blocks

A number of algorithms search for useful blocks and use them to create complete networks. This approach has achieved a lot of success in recent years with NASnet [45], ProxylessNas [37], hierarchical architecture search [46], and CoDeapNeat [47]. These algorithms have the advantage that each block has a smaller search space and is cheaper to evaluate than an entire network.

Other approaches used predefined blocks of layers such as [40, 41, 42], reducing the search space and can simplify the relationship between parts of individuals enabling simpler encoding strategies. This approach increases the reliance on domain knowledge from humans that is required in the algorithm. Approaches that search for blocks at the same time as complete networks, in comparison, require less domain knowledge but have a geater search space so will take a long time to search.

2.4.6 Genetic Programming and NeuroEvolution

Several studies [40, 41] are similar to the work that has been done in this project. [40] used Cartesian GP which creates an acyclic graph from the function sets which are modified between generations only using a mutation operator. Two function sets were used, one that contained only convolutional blocks and pooling layers and one that contained skip connection blocks and pooling layers.

One approach at using tree-base GP to design CNN architectures is GP-CNAS [41]. The terminals were made up of predefined blocks of layers and the function set defined the hyper parameters of the blocks such as the number of filters and strides. It also used dynamic crossover operation which favoured smaller changes in the second half of the generations. Consequently, it will favour exploration in the first half rather than exploitation in the second half. GP-CNAS is considered to be semi automatic NAS as the blocks of layers in the terminal set are decided by a human.

2.5 Summary

This section gives an overview into using CNNs for image classification. EC is introduced and the GP algorithm is explained in detail. NAS is then described in relation to CNNs. Designing complex CNN architectures by hand is difficult and automating the search of the architectures can make solving complex problems easier by requiring less domain knowledge and specialized skills. Relevant research around NAS and GP for image classification is outlined. While there has been a large number of approaches to using EC to do NAS there is no existing approaches that use tree-base GP.

Chapter 3

GPCNN

3.1 Chapter Goal

The first goal of this project was to develop a novel encoding strategy that can be used to represent complex CNN architectures while being compatible with crossover operations. This goal was achieved by using tree-based GP which allowed genetic operators to be applied to the neural networks as they were encoded as trees.

3.2 GPCNN

In the proposed method neural networks were encoded as trees. The genetic operators (mutation and crossover) both change whole subtrees which can result in large aggressive changes. However, representing the neural networks as trees rather than other encoding strategies means that the structure of the individuals are more similar to that of the final network. Using trees allows the implementation of skip connections [2] and densely connected blocks [48] to be generic rather than having a fixed architecture. Having more generic implementation of these ideas gives the tree-based GP algorithm the freedom to potentially explore more optimal solutions but this also increases the search space. Trees, however, are not able to represent the DAG structure of neural networks directly so an encoding strategy needs to be used.

The tree structure consists of leaves representing the inputs. The root of the tree represents the output of the CNN and is the Network node. There can be only one Network node per tree and it connects the parameters of the model. The two parameters of the network that are decided by the individuals are the architecture of the convolutional layers and what strategy will be used to flatten the output of the convolutional layers. All other parameters such as the optimization method, learning rate, early stopping patience, and training epochs are fixed for all individuals. Each tree has a backbone which is the branch with the node that represents the input to the CNN and controls the location of the connection of other leaves. These other leaves represent the input to a subsection of the graph. Each node has a context decided by its parents and children. Therefore, each node is aware of the nodes before and after it and can change its behaviour accordingly.

3.2.1 Function Set

The functional set contained 8 nodes:

- Layers

  - Convolutional node
  - Activation node
  - Pooling node
  - Batch Normalization node
  - Dropout node
- Connections
  - Skip connection
  - Densely connected
  - Auxiliary output.
  - Branching

The convolutional layers were represented by a convolutional node where the parameters for the number of filters and the size of the filters were represented as children that were enforced with Strongly typed GP. It also contained a parameter for how the convolutional node should handle the mismatching dimensions connections described by the Densely connected node. These connections were propagated up the tree using the node context. This was a choice between 0 padding, max pooling, and average pooling. The number of filters that could be chosen were 8, 16, 32, 64, 128 and the filter size options were 3x3, 5x5, 7x7 and 9x9. These form a wide range of values that are commonly found in handcrafted CNNs. Inputs to the convolutional layer was padded with zeros such that the size of the feature activation maps in the output of the layer was the same as input to the convolutional layer. Padding in this way means that the width and height dimensions are only changed by pooling and there is less likely to be mismatches of dimensions between branches.

The activation layer was locked to a Relu activation function as this is typically used by some successful deep CNN architectures such as ResNet. As the gradient is one when the value is above zero, therefore the weight updates are not scaled for each layer in the CNN.

The pooling layer used a parameter to distinguish between max and average pooling, chosen as these are the main ways of pooling. The pooling layers used a pool of size 2x2 and stride of 1x1. For each pooling layer the width and height of the activation maps were halved. The pooling layers were also set to do nothing if the dimensions of the feature maps were already less than 4x4, ensuring there is still sufficient data that applying convolutional filters is meaningful. The pooling layer had one parameter which determined whether it should apply max pooling or average pooling.

Batch Normalization was used as it is commonly used with convolutional and activation layers to normalize the data at certain locations in a neural network. It has been shown that normalizing the inputs to a neural network improves convergence [49]. Batch normalization is used at different places in the network to reduce Internal Covariate Shift and has been shown to have many positive effects on the training of neural networks, such as allowing the use of higher learning rates and regularizing the network making the network more generalized [14].

Dropout was also used as a possible way to regularize the neural networks [13] by outputs of some nodes that had a chance of being excluded by being set to zero. This changes the architecture of the network throughout the training, which regularizes the network as a more general network is learnt. However, dropout has the disadvantage of accuracy as test data will converge slower with higher levels of dropout but will converge to a higher level than a network with a low level of dropout. The selection criteria of the GP algorithm will therefore be biased to networks that have low levels of dropout that converge quickly but suboptimally compared with other individuals in the population.

The ResNet architecture [2] introduced the idea of skip connections which is where a point in the network is connected to a point earlier in the network. Doing this allows the loss to be propagated through the network without being distorted by weights near the output of the network. Skip connections are important for training deep neural networks due to the reliance of weight updates on the weights that are lower in the network as it adds a connection where there is no weights. Skip connections were implemented by a node with two children. The first child represents the network before the skip connection node. The second child represents the layers that are in parallel with the skip connection, as represented in Figure 3.1. The input node in the second child represents the output of the first child. The output of both children was concatenated together and used as the output from the skip connection node. Both the children of the skip connection node were required to contain a convolutional layer. The depth of the second child was limited to 5 nodes. This was to prevent networks from containing highly nested structures and forced the networks to be deeper rather than wider. Wide shallow networks can result in a large number of parameters but limit the ability of the convolutional filters to extract complex features. The power of convolutional filters in deep neural networks comes from operating on feature maps from the output of other convolutional layers and nesting the convolution operations.

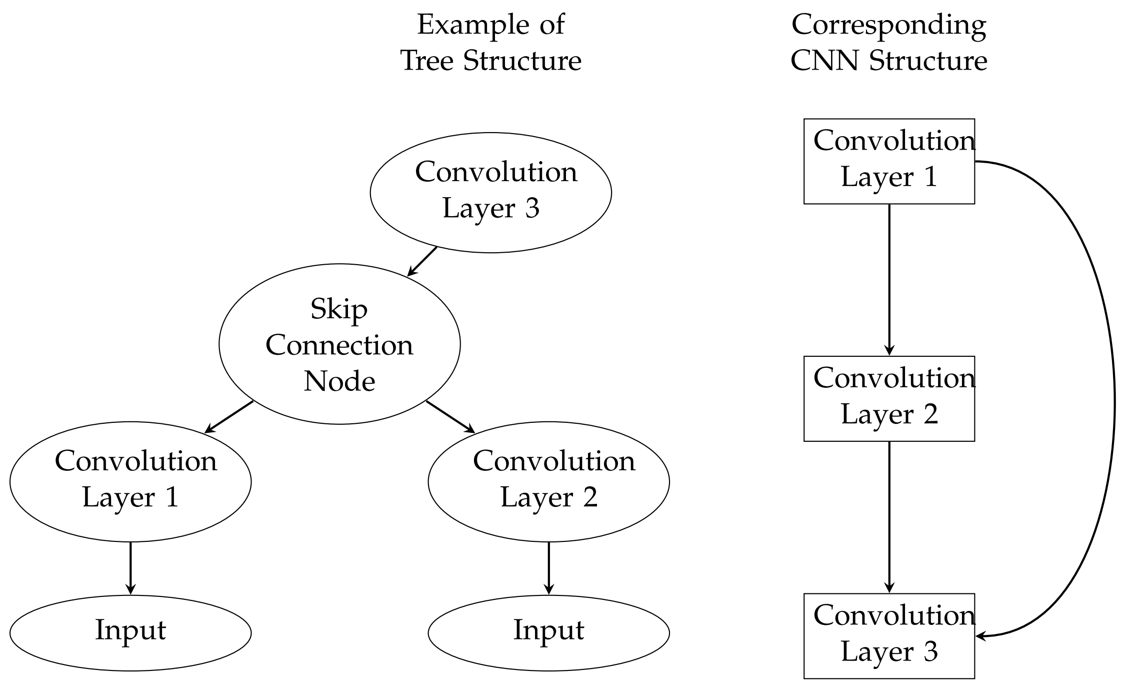


Figure 3.1: Example of a Skip connection node in an individual

Dense connectivity was proposed by [48] and is where the output of convolutional layers are appended to the input of following convolutional layers. It usually occurs in blocks that are made up of consecutive convolutional layers. But, this does not suit the tree-based GP as it cannot easily guarantee the consecutive layers of that type. As such the implementation of the dense connectivity blocks varies from the proposed method. The encoding strategy used a node where the input to that node is concatenated with the inputs in a number of following convolutional layers. The number of convolutional layers that are connected to this is determined by a parameter which is a number between 2 and 6. If the number of connections is greater than the number of convolutional layers in the rest of the network, then the dense connection node will have fewer numbers of connections to stop invalid numbers of dense connections. However, there can be a range of values that all create the same network. Connecting to the next *n* convolutional layers means that there can be large gaps between the convolutional layers that can contain other types of layers. The dense connectivity nodes can also work with other types of connections such as branches and and skip connections by using the node context. An example of this that only contains convolutional layers can be seen in Figure 3.2.

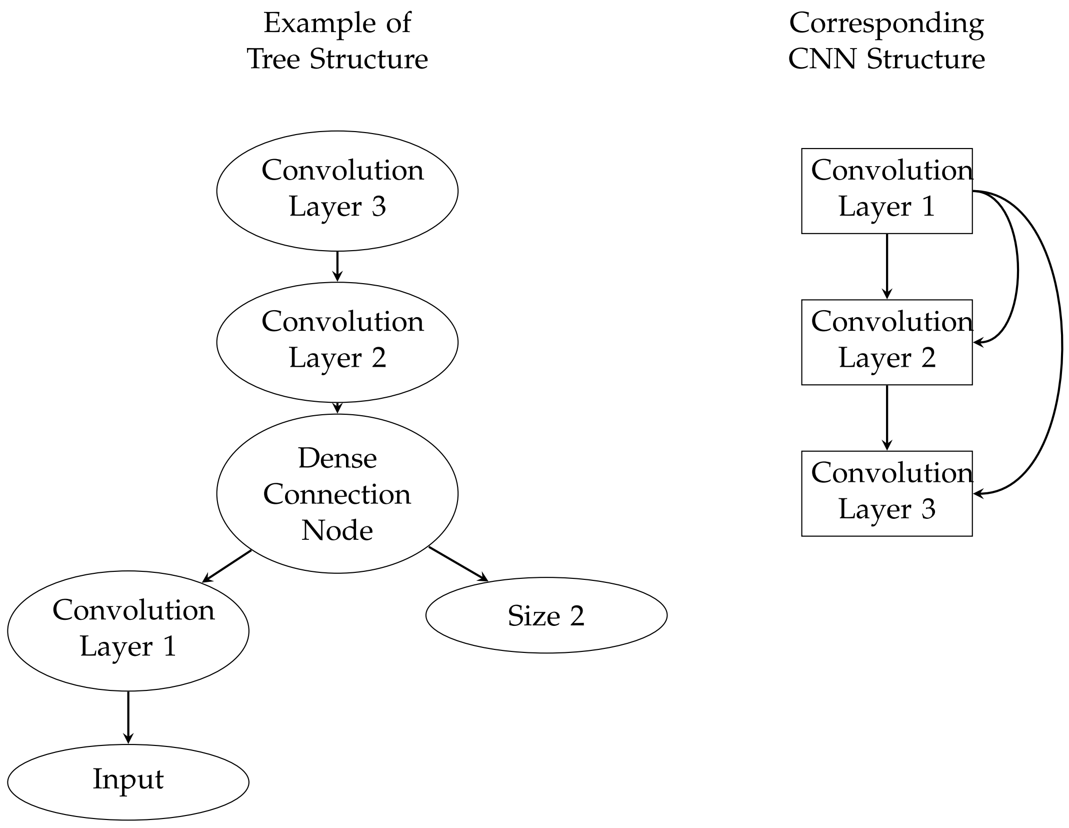


Figure 3.2: Example of a Dense node in an individual

GoogLeNet [20] introduced two notable motifs for neural network architecture. The first was Auxiliary outputs where a neural network has multiple outputs. The network is then trained through joint back propagation where the loss of the network is propagated from each output. Propagating the loss from each output means that layers close to the input layer are much closer to an output so there will be a more meaningful update to the weights. Due to back propagation using the weights of the layers, with random initialization, the updates to weights in layers far from the output are not meaningful and will not lead to learning a useful operation therefore inhibiting the neural network from training. Since Auxiliary outputs do not require much in the way of special connections they are easy to represent in the tree. They are represented by a node with a single child that passes the input to the next layer in the network. The output layers are connected to the input and added to the context and are handled by the root of the tree which is the Network Node. The structuring of this can be seen in Figure 3.3.

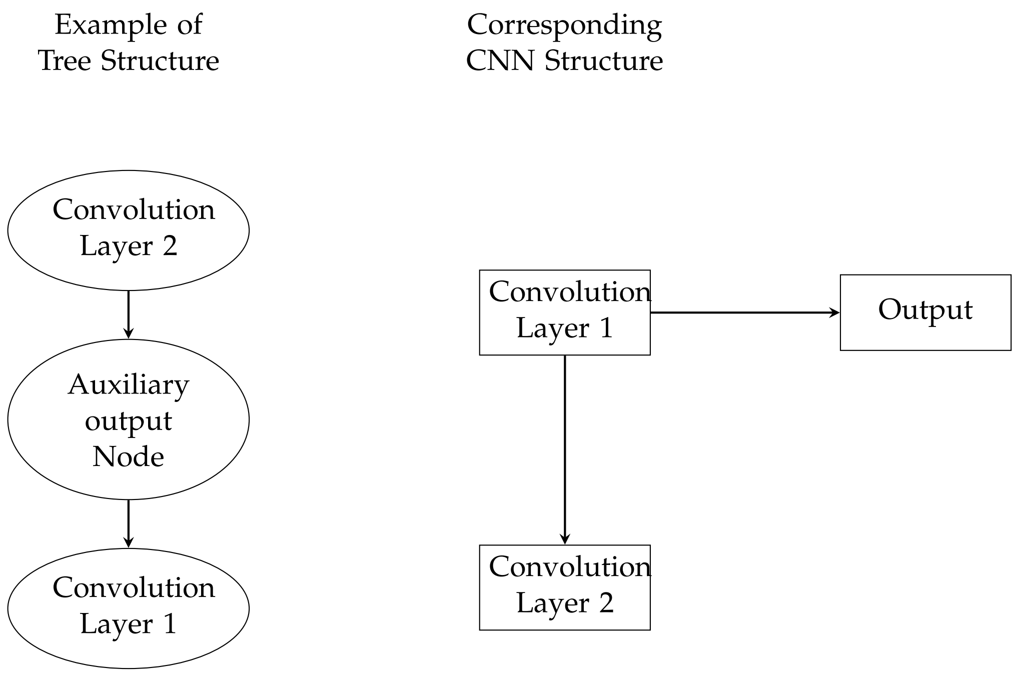


Figure 3.3: Example of an Auxiliary output node in an individual

The second motif that GoogLeNet used was inception modules. These are a number of layers such a 1x1 convolution (Feature Pooling), 3x3 convolution, 5x5 convolution and 3x3 max pooling layers that are all connected to the same input. The output of each of these layers is concatenated together to form the output of the inception module. To implement a general version of this in the tree-based GP encoding strategy, branching nodes were introduced which are nodes with 3 children. The first child represented the network before the branching node. The second and third child represented the two branches whose outputs were concatenated together to form the output of the branch node. Figure 3.4 gives an example of a branching. Branching nodes could be nested leading to more complex branching such as having more than two branches. However, the maximum depth of the branches was set to 5 to restrict the tree size and to avoid wide shallow networks. Both branches were also forced to contain at least one convolution layer. This was to avoid the cases where both branches did not perform any meaningful operation on the input and essentially only resulted in concatenating the input with itself a number of times.

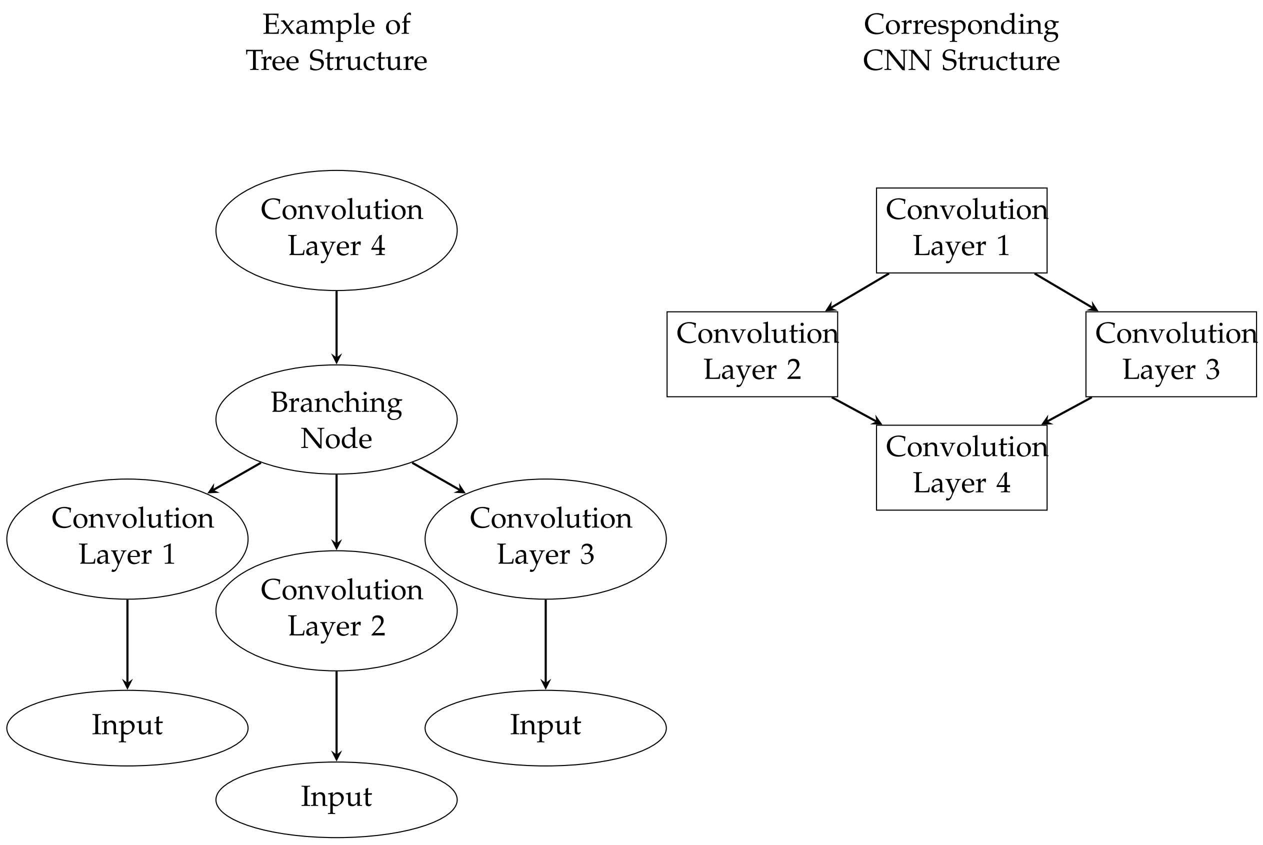


Figure 3.4: Example of a Branching node in an individual

The connectivity functions were designed to be generic so that a large number of possible DAGs could be represented by these functions. This leaves a significant area of the search space for the algorithm to explore and find knowledge that has not been used in handcrafted architectures despite the functions being inspired by handcrafted architectures.

3.2.2 Strongly Typed GP

By using Strongly typed GP it was possible to enforce that different branches contain at least one convolutional layer and was achieved by using two types: one that represents the state before a convolutional layer and the other representing the state after at least one convolutional layer has occurred. All the connection nodes were only able to be used after a convolutional node had occurred. This prohibits the case where some complicated connections without a convolutional layer causing the input image to concatenated with itself multiple times without feature maps being created. Concatenating the input with itself would increase the parameters and training time of the network without increasing the performance of the network. Branches from both skip connection nodes and branching nodes were enforced to contain at least one convolutional layer for the same reason.

3.2.3 Terminal Set

The terminal set for the tree-based GP algorithm was made up of parameters for the nodes and an input node. This input node represents the input into different parts of the network. For example, there was only one input node that represents the input to the entire neural network. For a skip connection, the input node in the right hand branch represented the input to the layers that were contained in the skip connection, which was the output of the left hand branch. This was similar to the branching node where the input nodes in the middle and right hand branches represent the output of the left hand branch.

Global pooling layers were also options, as well as a flattened layer, to transform the feature maps into a one dimensional form. Global average pooling was used in Class Activation Mapping and demonstrated that it could achieve similar accuracy with less parameters than the standard flatten operation when used on a ResNet on the ImageNet dataset [50]. Global pooling can result in far fewer parameters in the final fully connected layer due to the output dimensions being equal to the number of feature maps rather than $width \times height \times number$. Due to these successful results each individual had a parameter that defined the strategy to flatten the output of the convolutional section of the network chosen from a Global Max Pooling, Global Average Pooling or Flatten layer.

The tree-based GP algorithm was implemented in Python using the Keras and Tensorflow to train the neural networks and a heavily modified version of the DEAP framework for GP.

3.2.4 Fitness estimation

The fitness was estimated by training the CNNs for fewer epochs than is required to converge. This takes advantage of the assumption that more accurate networks will have higher accuracy at all epochs than less accurate networks. This, however, is not always the case due to overtraining where simpler networks are likely to have large increases in accuracy early on in the training due to the lower number of parameters to update. Due to the simplicity of the networks the accuracy will not increase past a certain level as the problem is too complex. This is different from a complicated network which will gain less accuracy for each epoch initially however it will have a higher max accuracy. Complex networks are also susceptible to overtraining where the accuracy on the evolutionary training set decreases while the accuracy on the gradient training set increases.

3.2.5 Voting

The GP algorithm creates a large number of individuals that do a reasonable job at classifying the images in the test set. Since evaluating individuals is computationally expensive only a small number of individuals were evaluated compared with the use of GP in other domains, such as symbolic regression. As only a small portion of the search space is explored there will be a high level of variability in the final population. Assuming that individuals that are diverse correspond to neural networks that have diverse behaviour, using a voting system that predicted class values based on the most commonly predicted class from a pool of the best individuals to provid better results than using only the best individual. Finding the best individuals to use in the pool of classifiers for the voting was done by starting with the classifier that achieves the highest accuracy on the evolutionary training set and then calculating how accurate this pool of one is on the evolutionary training set. Next, the classifier with the next highest accuracy on the evolutionary training set was added to the pool. This was continued until the accuracy of the pool started decreasing. This process can be seen in Figure 3.5.

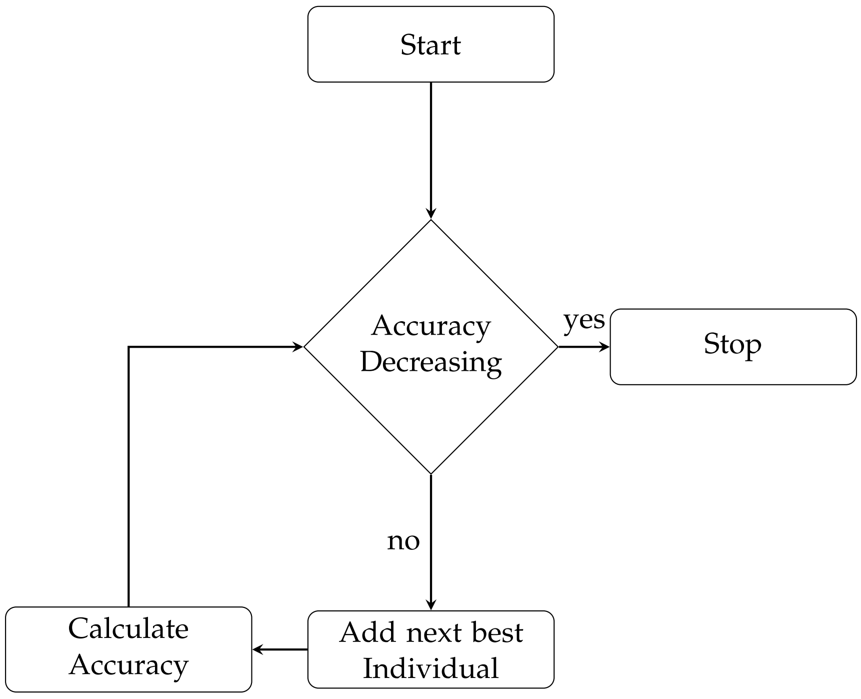


Figure 3.5: Process to find the best individuals to use for voting

3.3 Experiments Setup

[26] showed that performance of GP with high levels of crossover vs high levels of mutation performed differently. However, where the population size was large, crossover performed better than mutation. Unfortunately, due to the long time that is required to evaluate each individual the population size had to be kept small (50 individuals) and as such a high mutation rate was set compared with the crossover rate. The mutation rate was set to 65%, the crossover rate was 35% and the elitism rate was 10%. The mutation and crossover rate represents the chance that each individual will be selected in crossover or mutation. This means that an individual can be selected for both crossover and mutation in one generation. Using a higher mutation rate allowed for the diversity in the populations to be maintained as there will be a small amount of genetic material to choose from due to a limited number of individuals.

The algorithm was run for 20 generations and had 50 individuals in the population. There were three main reasons for choosing these values; the time that the algorithm would take to run; to balance the number of generations and population size so that there would be a reasonable number of individuals for crossover; and to ensure that there were enough generations so that the search space around fit individuals would be explored.

3.3.1 Dataset

The CIFA-10 dataset was used as the problem to test the GPCNN algorithm on. This dataset contains 60,000 instances split between 10 classes of images. Some examples of the class values are bicycle, car, cats etc. The dataset was split into three sections; the gradient training set; the evolutionary training set; and the test set. The gradient training set consisted of 45,000 instances and was used to train the CNNs that were generated from the individuals in the population. The evolutionary training set was made up of 5,000 instances. The split of 75/8/16 between the gradient training, evolutionary training and test set was chosen so that there was as much as possible in the gradient training set while still having an adequate number of individuals to assess the accuracy of the 10 classes of the CIFA-10 dataset. Stratified sampling was used so that the class ratios were all kept consistent. Fitness of the individuals was the accuracy on this dataset. The last dataset is the test set which was not used during the training process and was used only to evaluate the final solution at the end of the training.

3.3.2 Individual Training

During the evolutionary process the individuals were trained for 10 epochs on the gradient training set. After the GP search was completed, the individuals were trained for 60 epochs on the gradient training set. The weights of the network for an individual used in the final representation of an individual were those that maximised the accuracy on the evolutionary training set, rather than after 60 epochs had finished. This is to avoid the case where the loss function would evaluate to an undefined value and the network would no longer be able to be trained. All the training on the networks was done using Stochastic Gradient Descent (SGD) [51] with a learning rate of 0.01, momentum of 0, and weight decay of 0.

Two early stopping criteria were used when training individuals. The main form of early stopping was a patience criteria where the training was stopped if the accuracy on the evolutionary training set did not increase within a certain number of epochs [52]. There were two reasons for using this early stopping criteria. First, if an individual had an architecture that failed to learn any reasonable mapping between the images and labels then training was stopped after a small number of epochs thus saving a large amount of computational time. Secondly, to prevent overtraining of the individuals. An individual starts to become overtrained when the fitness accuracy decreases while the training accuracy increases for a number of epochs. In this case, the training was stopped to prevent the fitness accuracy from decreasing further.

Another form of early stopping was a timeout where the training was stopped if the network took too long to run. This was necessary to implement as some networks took substantially longer to run without accuracy improving. Timeout was set at 2 minutes per epoch so that individuals that were trained for 5 epochs had a time limit of 10 minutes. However, this was only checked at the end of each epoch so they were able to run past 10 minutes but a completed model was always made. The timeout of 2 minutes was chosen after running the algorithm and choosing a level that would only stop networks that took significantly longer than the usual networks. A 2 minute timeout had a small impact on the overall fitness of the population while stopping the networks that took an extremely long time (times of up to 7 hours for one individual were observed). Due to the way memory is allocated, if the network took up the majority of the memory on the GPU, the data had to be continually loaded causing the training time to sharply increase if models started to approach the memory limit of the GPU.

Small changes to parameters such as the number of filters, filter size, and dropout rate are unlikely to cause large changes to the final accuracy of the network. A set of values were chosen for the parameters rather than numerical ranges and consequently small changes in the parameters were not possible. However, this decreases the amount of exploitation the algorithm could do and therefore the algorithm would converge faster as it spent less time evaluating trees that were marginally different from existing trees.

During the evaluation of the algorithm, the diversity of the population was measured. The measure used was the distance metric proposed by [53]. This method of calculating the distance between two trees adds dummy nodes to a pair of trees until the structures of both match. Then the difference can be calculated by assigning a difference between each possible pair of nodes and sum the difference values for the whole tree. The diversity in the population was measured by the average of the distances between the trees through the whole population. An advantage of this distance metric is that larger trees will be more likely to be further apart. A disadvantage is that there can be trees with similar nodes but offset by one or branch ordering and therefore have a large diversity when they are similar.

[14] found larger batch sizes worked better with SGD on CIFAR-10 and MNIST datasets. They used batch sizes ranging from 16 to 1024 and found that using a batch size of 1024 achieved the highest accuracy on both datasets. The drawback to using a larger batch size is the increased memory requirement of the GPU. Therefore, in this project, smaller networks were able to use a larger batch size whereas larger networks needed to use a smaller batch size to be able to be run on the GPUs available. The batch size for each network was started at 1024 and if the GPU ran out of memory the batch size was halved until either the network was able to be trained or the batch size got smaller than 32 where the network was deemed untrainable and the fitness value was set to -1.

3.4 Results and Discussions

Table 3.1: Accuracy on Test set

	Accuracy at 10 epochs	Accuracy at 60 epochs	Accuracy of voting
Run 1	68.4%	76.7%	81.3%
Run 2	71.3%	70.7%	79.5%
Run 3	70.4%	82.0%	82.1%
Run 4	74.4%	78.1%	84.0%
Run 5	72.5%	76.0%	79.1%

Table 3.2: Average diversity of top 5 individuals

	Average Diversity
Run 1	20.4
Run 2	99.5
Run 3	91.2
Run 4	29.6
Run 5	25.7
All Runs	258.55

The tree-based GP algorithm was biased towards models that reached high accuracy in under 10 epochs regardless of how much the accuracy will increase after 10 epochs. This caused the increase in accuracy between 10 epochs and 60 epochs to be small at 5.3% on average.

Voting significantly increased the accuracy of the algorithm. The average accuracy of the top individual was 76.7% while the average accuracy from voting was 81.2%, indicating that there is significant diversity in the classification outputs of the neural networks created by the best individuals. Small changes made to the trees and neural networks produced a large amount of diversity in the classification outputs as the diversity of the best individuals was very small compared with the diversity of the rest of the population. The diversity between the different runs of the algorithm was also very high, indicating that the starting population of the algorithm had a large effect on the outcome of the algorithm at 20 generations. It was expected that running the algorithm for more generations would decrease the diversity between runs as the different runs would overlap more as they searched throughout the search space. However, this did not occur in practice because of the small portion of the search space that was evaluated. Only 655 individuals were evaluated on average which is extremely small compared to the total search space.

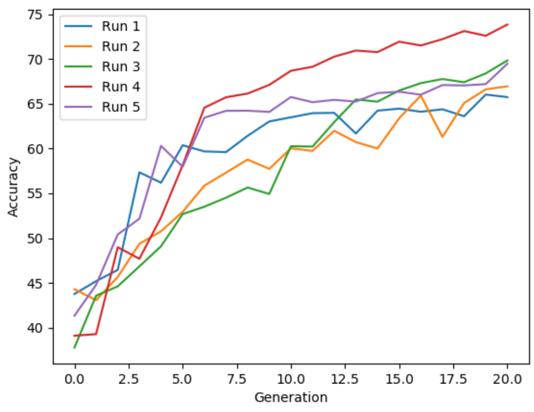


Figure 3.6: Accuracy on test set of top 5 fittest individuals

None of the five runs of the algorithm converged to a local minimum. The lack of convergence is shown in Figure 3.6, where the accuracy of the top individuals in each generation steadily increases over the 20 generations. Lack of convergence indicates that there is further potential to learn better classifiers. However, this would take significant computation time. For example, a single run of the algorithm was done for 30 generations, had a population size of 150, and each individual was trained for 30 epochs. These parameters resulted in the best individual achieving 89% accuracy on the test set after being trained for the full 60 epochs. However, this run took 15 days to complete.

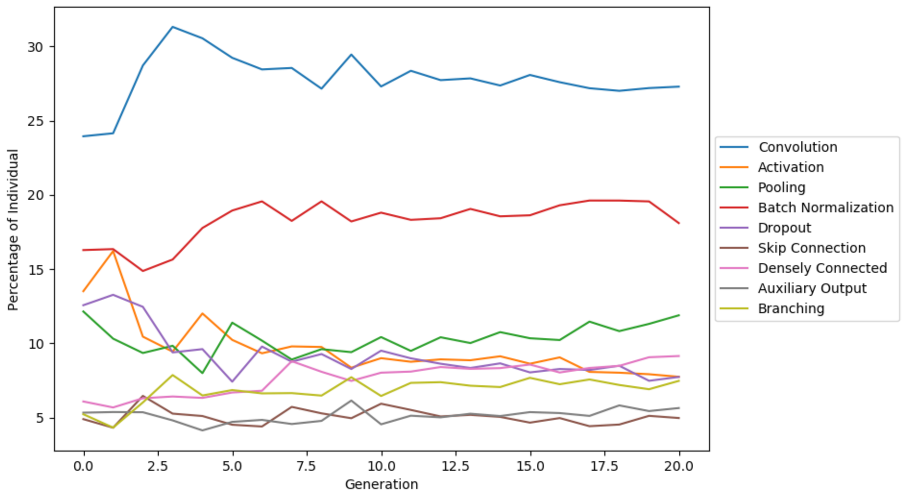


Figure 3.7: Composition of 5 fittest individuals each generation

The difference in the composition of the fittest 5 individuals during the evolutionary process was small as seen in Figure 3.7.The densely connected and branching nodes became more common the more generations were evaluated. The nodes that decreased the most were the activation nodes and the dropout nodes, showing that networks with more complex connectivity performed better than networks that were more sequential. The decrease in activation and dropout nodes was expected as activation layers with no convolutional layer between them is equivalent to having only one activation layer and networks with too much dropout train slowly.

3.5 Chapter Summary

This chapter outlined the base GPCNN algorithm which used a tree-based GP to evolve CNN architectures.

The results that the base GPCNN achieved on the CIFAR-10 dataset shows that it is a viable approach to automating the design of CNN architecture and can be used as a baseline for further improvements. The base GPCNN, however, did not perform well compared with existing state-of-the-art algorithms, as expected, being the first research in this area. There are two causes for the lower performance. The first, that the base GPCNN was not run for as long as the existing algorithms. The GPCNN showed that after 20 generations it had not reached a local minimum. However, it was still making steady improvements. The second reason, related to the first, is that the search space for the GPCNN was very large due to the use of trees for encoding the CNN architectures.

Chapter 4

GPCNN with new Partial Subtree Crossover operator

4.1 Chapter Goal

The goal of this chapter is to improve the crossover operator that the base GPCNN uses to increase its flexibility so that it can exploit higher fitness individuals. This can be done by allowing the crossover to make small changes on any part of the individual within one generation rather than requiring it to take several generations to make small changes. Allowing small changes to individuals will enable GPCNN to evaluate the search space close to individuals more easily.

Traditionally, the crossover operator used with GP exchanges complete subtrees. Exchanging complete subtrees means it is difficult to make changes in the middle of the individuals without making large changes from the root of the subtree to the leaves. Consequently, the algorithm will struggle to explore the area around individuals with high fitness values due to the large changes that have to be made to change nodes that are far away from the leaves of the individuals.

To allow searching of the space surrounding an individual, nodes in the individual have to be able to be changed without changing the entire subtree from the node. One method would be to use a mutation operator that would mutate single nodes, replacing them with a random node that has the same signature. A signature is the number and type of children and the type of the node which can apply to a single node or a partial subtree. However, since the parameters of nodes are part of the subtrees there are only a limited number of nodes that share similar signatures. To allow for more flexibility in the operator it is proposed to use a new partial subtree crossover operator.

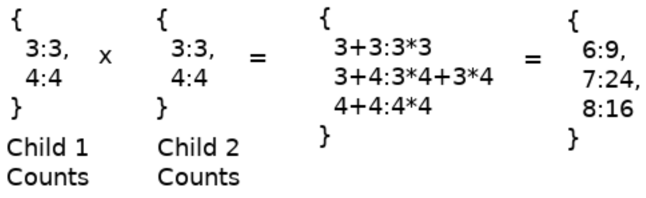
4.2 Partial Subtree Crossover

The partial subtree crossover operator proposed uses a modification of the traditional crossover operator. It swaps two random partial subtrees which have the same signature in the parents. Swapping partial subtrees allows nodes in the individuals to be changed without changing the parents or children of the node.

One potential issue with using partial subtrees is the vast number of partial subtrees that are possible for the typical size of the individuals that perform well on the CIFAR 10 dataset. The number of partial subtrees in a binary tree scales at a rate of 10^{2^d} where d is the depth of the tree. Iterating through all these partial subtrees would be infeasible. However, a random signature can be chosen and then two partial subtrees that match these signatures can be extracted using little computation time.

The partial subtree crossover operator chooses a random signature by using a weighted random choice on the sum of the frequencies of subtrees that had the signature in both parents. Signatures that only appeared in one parent were ignored. It is simpler to understand how the operator chooses a random partial subtree if types from Strongly Typed GP are not considered. Then signatures can be represented by the number of connection points rather than the type of the root and the counts of the different types of connection points. Trees are recursive data structures therefore signatures that use a node as the root can be counted by counting the signatures of the node's children and combining them. These counts can then be aggregated for all nodes in the tree to generate the counts for the signatures of any root.

In the case where there is only one child, the only operation needed is to increment the counter for the signature with one output. However, as exemplified in Figure 4.1, where there is more than one child the counts of the children can be combined by multiplying the counts and summing the signatures from each child with the counts from all the other children. Calculating the counts for a tree can be achieved with Algorithm 1.



```
{             {             {                    {
  3:3,    x     3:3,    =     3+3:3*3        =     6:9,
  4:4           4:4           3+4:3*4+3*4          7:24,
}             }               4+4:4*4              8:16
Child 1       Child 2       }                    }
Counts        Counts
```

Figure 4.1: Example combining counts from children

In Figure 4.1 each item in Child 1 Counts is combined with each item in Child 2 Counts. When there are two items with the same signature the counts are summed, representing that each partial subtree in Child 1 can be combined with any partial subtree in Child 2 to form a new partial subtree.

Algorithm 1: Count the signatures in a tree

```
1  CountSignatures (R)
2      if ||children|| == 0 then
3          return {0:1}, {0:1} // there is only one possible subtree with 0 connection points
4      end
5      if ||children|| > 1 then
6          carry = {} child_aggregation = {}
7          foreach child in children do
8              carry = combine_child_counts(carry, CountSignatures(child)[0] + {1:1})
                 //Combine the counts of all the children with the extra one subtree that contains
                 only the root node.
9              child_aggregation += CountSignatures(child)[1] + {1:1}
10         end
11         return carry, child_aggregation + carry
12     end
13
```

In Algorithm 1 Line 3 handles the case where the root is a leaf so the only possible partial subtree is the subtree that contains only the root. The algorithm has two return values because to calculate the counts of the subtrees that have a particular node as the root, the partial subtree counts of only the immediate children are needed, however, the final result of the subtree is the aggregation of this value for all nodes. Line 8 combines the counts of all the children with the extra subtrees that contain only the root node.

By using strongly typed GP, the signatures become the type and count of the connection points. The type of the input can be ignored while calculating the counts for one node as it is constant. However, the type of the input needs to be included in the signatures when aggregating the counts of each node.

After the counts have been obtained, they are filtered to the signatures that appear in both parents. An input and output signature is randomly chosen with a weighted random choice based on the counts. Then the root node can be found by choosing a random number from the count for the signature and using depthwise iteration over the tree, continuing until the sum of the subtrees at each visited node becomes larger than the number chosen. The nodes that represent the connection point can be randomly chosen by choosing leaves equal to that of the signature. To make a subtree into a partial subtree, a random point between each chosen leaf and the lowest common ancestor from any other chosen leaf is chosen to be a connection point in the final partial subtree. This does not work with most strongly typed GP trees. However, it does work in the case when the type does not revert and is guaranteed to change to all the other types between all the leaves and the root. This is taken advantage of in this project.

4.3 Experiments Setup

The hyperparameters of the algorithm were not changed from the parameters of the basic GPCNN. The only difference is the new crossover operator.

To reduce computation time, the parameters of the nodes were compressed into the nodes. Therefore, the crossover operator would not exchange subtrees that only consisted of one parameter and would always include a change to the architecture of the network. This makes mutation the only operator that is able to change a layers parameters. Since compressing the parameters into the nodes means that there are only two types in the trees (before any convolution layer and after any convolution layer), the trees meet the requirements to efficiently extract a partial subtree given its signature as described in Section 4.2.

4.4 Results and Discussions

The size of the subtrees that were exchanged by crossover were much larger using partial subtree crossover than traditional crossover. This is because nodes that are close to the root of the tree have a greater number of partial subtrees causing these nodes to be chosen as the root of the subtree more often than nodes closer to the leaves of the tree. Usually trees have more nodes close to the leaves rather than close to the root due to the branching in the tree. Consequently, traditional crossover is more likely to choose a node close to the leaf than partial subtree crossover would.

The partial subtree crossover was able to make more meaningful exchanges than traditional crossover but was less likely to do so. On average, the fitness value of a descendant was 13.1% lower than the average of its parents when using partial subtree crossover compared with 5.0% for traditional crossover and 7.2% for mutation. This shows that on average partial subtree crossover performed worse than replacing subtrees with a randomly generated subtree. However, when the descendant had a higher fitness value than the average of its parents, partial subtree crossover had a larger increase than traditional crossover (6.7% on 26.2% of descendants compared to 4.8% on 32.6% of descendants respectively). The reason for the increase in destructiveness is that the size of the subtrees exchanged by the partial crossover operator was significantly larger than traditional crossover. On average partial subtree crossover exchanged subtrees of 34 nodes, whereas, if traditional crossover had been used only 12 nodes on average would have been exchanged as seen in Table 4.1.

Table 4.1: Average size of subtrees that would have been exchanged

	Partial Subtree Crossover	One Point Crossover
Run 1	53	15
Run 2	28	12
Run 3	40	12
Run 4	24	9
Run 5	23	8

Table 4.2: Accuracy on Test set

	Accuracy at 10 epochs	Accuracy at 60 epochs	Accuracy of voting
Run 1	76.5%	84.1%	86.5%
Run 2	72.5%	80.4%	83.0%
Run 3	69.1%	76.1%	79.8%
Run 4	68.7%	77.0%	80.0%
Run 5	71.0%	76.8%	81.3%

Using partial subtree crossover did not significantly increase the accuracy of the GPCNN algorithm with an average accuracy of 82.2% compared to the base GPCNN which had an average accuracy of 81.2%. Partial subtree crossover was more destructive but made better improvements. These effects cancelled each other out, leading to a very small change in the performance of the algorithm. Accuracy between individuals after 10 epochs of training and 60 epochs increased more with the partial subtree crossover algorithm: 3.9% compared with 2.6% for traditional crossover. Although this difference in accuracy was not significant, the individuals in the search for an optimal architecture were not as biased towards individuals that trained well after 10 epochs and did not overtrain afterwards as was the case for the base GPCNN.

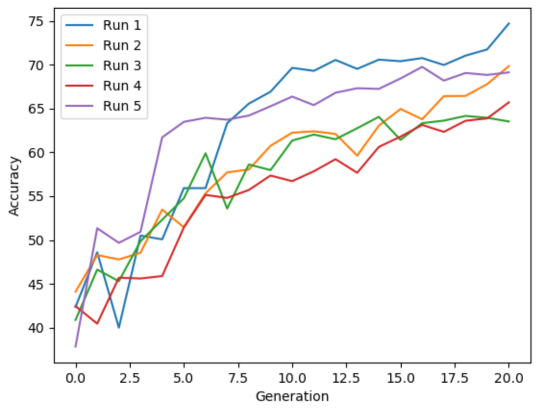


Figure 4.2: Accuracy on test set of top 5 fittest individuals

After 20 generations neither the GPCNN with the partial subtree crossover nor the base

GPCNN had converged. This can be seen by the high gradient in the Figure 4.2. This is expected as using the partial subtree operator means that it takes longer to converge as it is able to exploit locations of high accuracy better. However, due to partial subtree crossover being much destructive on average than traditional crossover, the chance it will exploit a region will be smaller so it will start converging sooner but finish converging later. This can be seen by the slowing of the increase in accuracy of the fittest 5 individuals. In the last 10 generations the accuracy of fittest 5 individuals had a gradient of 0.55 with base GPCNN compared with 0.53 when using the partial subtree crossover operator. To know which effect dominates, the GPCNN with the partial subtree crossover would have to be run for longer.

The same starting populations were used for the runs of GPCNN with the partial subtree tree crossover and with traditional crossover. However, the networks used a different initialization points leading to different fitness values in the first population. Therefore, the fitness values would change each time they are evaluated further increasing to the inaccuracy of the estimation of the accuracy of the individuals.

4.5 Chapter Summary

This chapter outlined a new crossover operation which was then applied to the GPCNN algorithm. The advantages of this operator is its ability to efficiently search the space around an individual. However, the results suggest that the crossover was more destructive on average than existing crossover operators and with the low number of generations was not able to fully show the benefit of it. Any benefits of being able to search around an individual were hidden by how destructive the partial subtree crossover operator was.

Chapter 5

GPCNN with Variable Fitness Evaluation

5.1 Chapter Goal

The goal of this chapter is to improve the selection operation of the base GPCNN by using Variable Fitness Evaluation to detect stop inaccurate estimations of fitness values causing the GP algorithm to become misguided.

Training an individual to its full potential is a lengthy process and it is difficult to determine the optimal number of epochs needed to train each individual. A faster approach to training individuals fully would be to estimate the accuracy of the individuals rather than determining their optimal accuracy.

There are several ways the base GPCNN implicitly manages bloat. One way is by limiting the number of epochs that each individual is trained for. Larger networks take longer to train due to a large number of weights, so the weight space has higher dimensionality and is harder to traverse with gradient descent. With gradient descent, updates to weights are affected by weights from layers closer to the output. Deep networks also take longer to train as the updates to weights close to the input are affected by the random weights close to the output. This effect is significant because the weights of layers close to the input significantly affect the accuracy of the network as the following layers rely on the output of these layers. The slower training of large networks leads to a lower accuracy and fitness value after training for a small number of epochs regardless of the overall learning potential of the individual.

Another way the base GPCNN implicitly manages bloat is by limiting the depth of the trees for each individual. Nodes that help the individual and nodes that only contribute to bloat have to fit within the max depth. The GPU memory limit also serves a similar role by restricting the number of parameters in the networks as networks that do not fit on the GPU memory are unable to be trained and are given a negative fitness value.

Although bloat is implicitly handled by the base GPCNN, handling it explicitly will reduce the risk of having bloat in individuals. Bloat in individuals can be detrimental to the algorithm due to the ease with which bloat can increase the time required to evaluate an individual and thus cause the runtime of the algorithm to increase.

5.2 Variable Fitness Evaluation

To offset the fall in accuracy by the use of estimations the most accurate individuals are trained for more epochs. This verifies that that network will become more accurate if it is trained for longer. Verification can be done on a number of the best individuals in a population for each generation. The accuracy values, which are achieved by training the most accurate individuals for longer than the other individuals, will be higher on average. This creates a problem with using elitism as it becomes very hard for new individuals to be added to the section of the population that has been trained for longer, pushing the algorithm into a local minima very quickly. To avoid this, individuals that have already been trained for longer are excluded from consideration so that a fixed number of individuals will be trained for the greater number of epochs each generation. This way they do not take over the population as they are already the fittest individuals and in each generation and many new individuals are added through crossover and mutation that have not been trained for a large number of epochs. In essence, using a fixed number will be training some of the best new individuals, created by mutation and crossover, per generation to decide whether they are added to the group of individuals that have been trained for longer.

To explicitly control bloat in the individuals a double tournament was used. The first tournament chose a number of individuals based on the fitness of the individuals. The second tournament selected the individuals with the lowest time taken per epoch. This was used because the reason for controlling bloat was to keep the training time low rather than to have simpler networks. While size is a good estimator of the complexity of the network it is not as good at estimating the training time due to different architectures have varying numbers of parameters causing different training times for individuals with the same size. It is common to do the second tournament on the size of the tree [28]. However, in this application, the number of nodes would not be an accurate representation of the size or complexity of the network that it produces due to different layers contributing different amounts to the network. Examples would be the connection nodes and the pooling, and batch normalization layers not adding as much complexity as the convolutional layers.

5.3 Experiments Setup

In the double tournament, the first tournament chose the 2 fittest individuals from 4 randomly chosen individuals which meant the second tournament had a 75% chance of choosing the network that ran faster and a 25% chance of choosing the slower network.

The top 5% of the population that had only been trained for 10 epochs, was trained for 40 epochs. Training for 40 epochs was completed without any early stopping so that individuals that started to become overtrained, continued to be trained and the fitness of these individuals was harmed. This was advantageous as it helped to remove individuals from the population that were susceptible to overtraining. There is almost no risk that there is an individual that does not change gets trained for 40 epochs as it would have to be one of the fittest individuals that have been trained for 10 epochs.

5.4 Results and Discussions

Variable fitness evaluation pushed the GP algorithm into a local minimum much faster than other approaches did. GPCNN with variable fitness evaluation had mostly converged within 20 generations. The diversity in the top individuals was also much smaller by the 20th generation compared with the first generation. Since the best individuals were already trained for 40 epochs, training them for 60 epochs marginally increased the accuracy of the best individuals. Voting also did not significantly increase the accuracy from the accuracy of the best individuals due to the lack of diversity in the best individuals at the end of the evolutionary process. The combined effect of training for 60 epochs and using voting had a

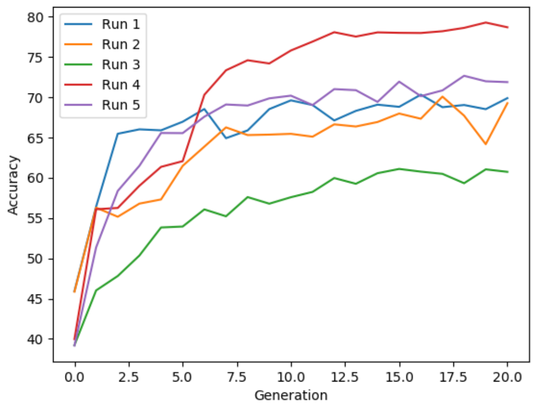


Figure 5.1: Accuracy on test set of top 5 fittest individuals

statistically significant increase in accuracy compared with the accuracy of the individuals while optimizing the architecture.

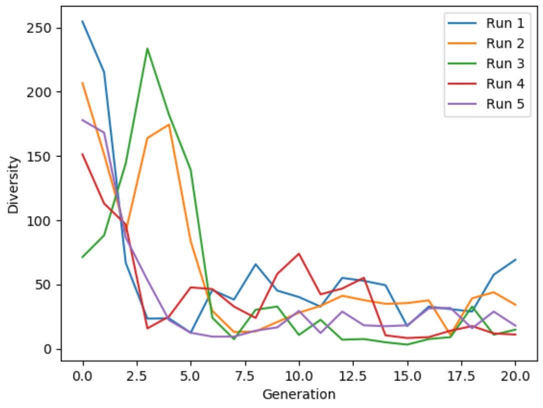


Figure 5.2: Diversity of 5 fittest individuals

Table 5.1: Accuracy on Test set

	Accuracy after search	Accuracy at 60 epochs	Accuracy of voting
Run 1	77.7%	78.0%	83.6%
Run 2	78.0%	78.2%	79.3%
Run 3	68.7%	73.6%	73.6%
Run 4	81.7%	81.1%	85.5%
Run 5	76.3%	76.4%	80.3%

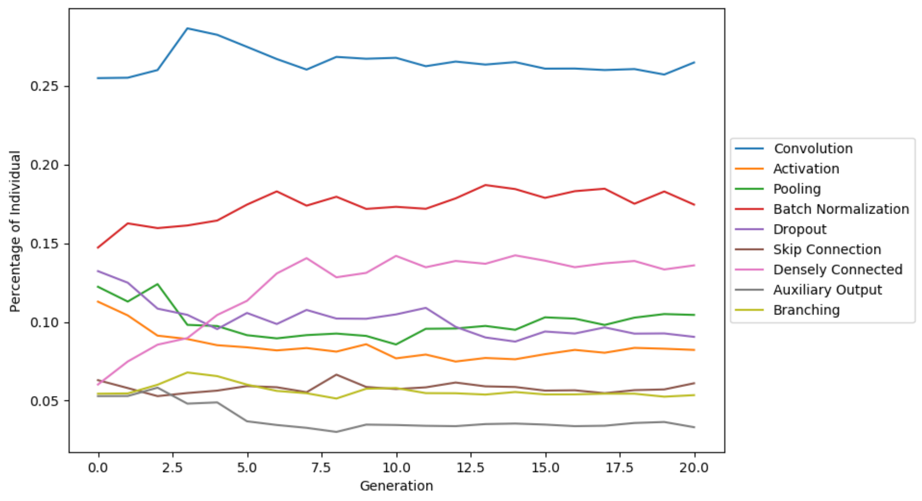


Figure 5.3: Composition of 5 fittest individuals each generation

After the first 5 generations there was almost no change (as shown in Figure 5.3) in the composition of the individuals due to the low diversity of the population. Activation and dropout nodes became less common in the first 5 generations which is similar to the base GPCNN. Branching nodes did not significantly change while the densely connected nodes become much more common. This shows that densely connected nodes perform better when trained for more epochs. Whereas branching nodes do not perform as well when trained for a larger number of epochs.

The average time taken per epoch for the training of each individual was smaller with the use of the double tournament, achieving an average of 54.6s/epoch vs 71.0s/epoch with the base GPCNN. Training the individuals for more epochs put pressure on increasing the time per epoch as the selection operator would favour larger networks that trained slower. The time per epoch when using variable fitness evaluation was more variable than with the baseline GPCNN because of these opposing forces.

The accuracy of the best individual on the evolutionary training set increased by 8.5% from the 10th epoch to the final accuracy. Base GPCNN had an increase in accuracy of 7.9%. While this was not statistically significant it represents how the individuals in the base GPCNN were biased the accuracy of the individuals at 10 epochs rather than how much they would have continued to train for.

5.5 Chapter Summary

This chapter proposed a way to ensure that the GPCNN algorithm was not misled by the fitness estimations called variable fitness evaluation. This was done by finding accurate fitness values for individuals that showed that they probably had a high fitness value. Using variable fitness evaluation did not significantly change the accuracy of the algorithm. The low diversity of the best individuals in the population showed that the algorithm was much closer to converging than the base GPCNN and therefore that the total learning ability of the algorithm was lower.

Chapter 6

GPCNN with new Building Block Initialization for Initial Population

6.1 Chapter Goal

The goal of this chapter is to propose a way to include domain knowledge from research of image classification algorithms into the base GPCNN. This can improve the performance of the initial population allowing the algorithm to converge after less generations.

Domain knowledge can be incorporated into GP in multiple ways. One way is with the function set. Domain knowledge is included by using functions that are known to work well in the domain. For the case of GPCNN this would be layers choice and the range of the parameters of the layers. If functions or parameter values are included that add no benefit to the final network then they will have a negative effect on the rate of convergence of the algorithm due to the increase in search space. It is hard to know what functions are needed to solve a problem without having already solved it. While convolutional operators perform well on image classification, it is possible that a new operator that has not been thought of will perform better.

It has been shown that choosing a better initial population can improve the convergence and final result of an evolutionary algorithm [54]. There has been some research into ways to choose a better initial population. These range from using opposition based learning [54], different diversity measures [55], and using an estimate from a different part of the algorithm [56].

Extensive research has been done in the domain of image classification. This can be seen as areas of the search space that have already been explored which can allow the algorithm to converge quicker, reducing the number of generations needed to run the GP algorithm for before finding individuals with high levels of fitness. However, it will be more difficult for the algorithm to find new structures and knowledge about the problem domain as it will be trying to build on existing knowledge rather than starting from scratch.

There are a number of datasets that have similar properties to the CIFAR-10 dataset such as the CIFAR-100 [36] and ImageNet [57] datasets. There is a large number of algorithms that perform well on all of these datasets when using transfer learning such as Gpipe [7] that was trained on ImageNet and achieves state-of-the-art accuracy of 99% on CIFAR-10. EfficinetNet [9] that used the same approach and achieved 98.9% accuracy, demonstrating there is a large amount of similarity between image classification problems. So the domain knowledge from one problem can be applied on another problem and expect a reasonable level of accuracy. These approaches transferred the weights of the network from one domain to another this is a much less general than transfering just the architecture. Transfering the weight of a network requires the classes in the new domain to be classified by similar features from the previous domain.

One way domain knowledge is already included in the base GPCNN algorithm is the different types of layers that make up the function set and the range of paraments for the layers. However they are designed to be generic and contain as little assumptions as possible so that GPCNN will be applicable in a wide range of domains.

6.2 Building Block Initialization

To include the domain knowledge of the algorithm the initial population can be built using a function set made up of building blocks. The building blocks in the initial functional set represent partial subtrees. These partial subtrees are made up of functions from the function set used during the evolutionary process that have been found to be useful from existing research. This method of initializing individuals does not have to be used for the entire initial population as a portion of the population can be initialized using a different initialization technique to maintain diversity.

In Chapter 2 approaches that searched for building blocks and used predefined building blocks were outlined. This approach sits in between these methods of using building blocks as the connectivity in the blocks can still be modified later in the evolutionary process. Using building block initialization will not affect the search space of the base GPCNN algorithm as their are no restrictions placed on the evolutionary process after the initial population.

The functional set that was used was the same as the functional set in some similar research which used an evolutionary approach to CNN architecture design for image classification. They are AombaNet [1], Gpipe [7] and hierarchical architecture search [46]. These papers achieved either state-of-the-art results or nearly so. Therefore, three types of convolutional layers were used: separable convolution layers, depth wise separable convolution layers and normal convolution layers.

6.3 Experiments Setup

Other than changes to the initialization and the function set all other parameters stayed the same as the base GPCNN.

6.3.1 Building Blocks

The building blocks that were used are:

- Convolutional block
- Skip Connection block
- Dense block

The convolutional block outlined in [14] consisted of a convolutional layer followed by a batch normalization layer then an activation layer. These layers have been shown to work well with each other in various configurations. The order of these layers is the same as the block in CGP-CNN [40] and is commonly used in hand-crafted architectures [17, 48]. Figure 6.1 shows the partial subtree that the convolution block will represent in the individuals.

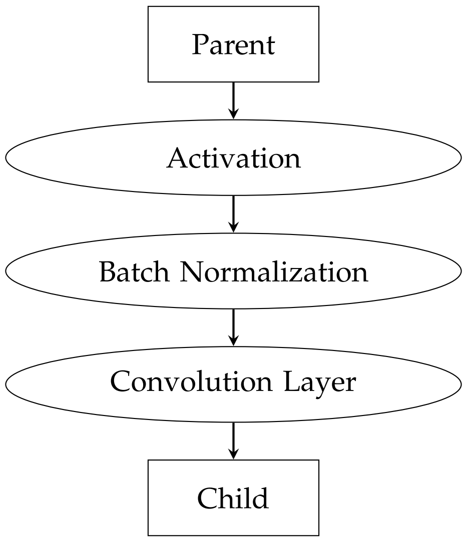


Figure 6.1: Convolutional Block

The skip connection block consisted of a skip connection over two convolutional layers. The subtree can be seen in Figure 6.2. This is consistent with the ResNet block that was proposed by [17] and achieved 93.57% accuracy. These styles of blocks have been successfully used in many different configurations with many different modifications such as using an exponential linear unit activation function [10], changing the order of operations to batch normalization, activation then convolution [11], and increasing the number of filters the further the block is from the input of the network [12]. These modifications all achieved above 93

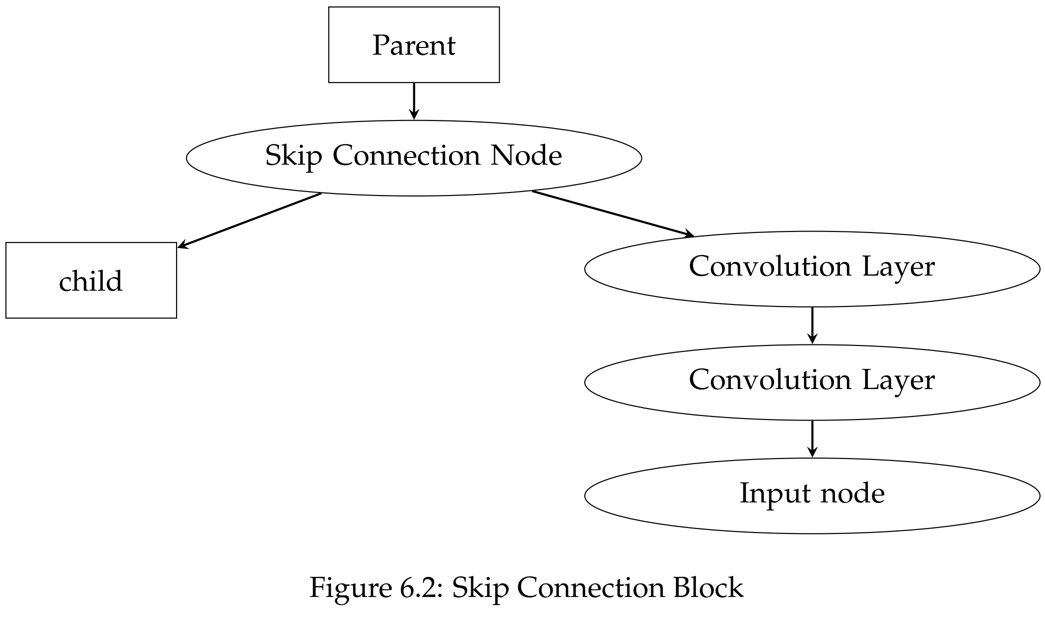


Figure 6.2: Skip Connection Block

The dense block used followed the dense block that was outlined by [48] and consisted of four convolutional layers that were densely connected. This DenseNet architecture was used to gain 96.54% accuracy on the CIFAR 10 data set. The subtree architecture is shown in Figure 6.3.

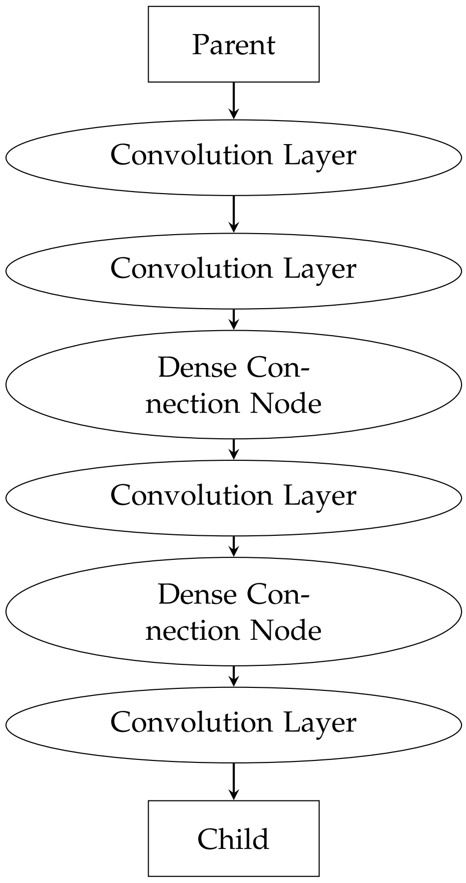


Figure 6.3: Dense Block

6.3.2 Initilization

Diversity in the initial population was maintained by dividing the initial population in half. One half was initialized using only the convolutional block, skip connection, and dense building blocks. The other half was generated using the standard Ramped Half-and-Half method [24]. It was necessary as diversity amongst the individuals in the initial population generated by the building block initialization was very low.

Since the building blocks can represent multiple layers, the depths of individuals created by the building blocks was controlled by a separate depth limit.

6.3.3 Function Set Adjustments

Separable convolution layers were added to the functional set. These were added due to the successful application of them by Amoeba net [1] and then GPipe [7]. Depthwise separable convolutional layers were added due to [58] concluding that they can be used in the place of inception modules.

Feature pooling used in GoogLeNet [20] was also added as a different type of pooling. This was achieved by using convolution layers with filters of size 1x1 which results in the width and height of the feature maps not changing. The number of filters was set so that the output of the layer had half the number of feature maps as the input to the layer. If the number of feature maps is smaller than five in the input to the layer, then the pooling node is ignored and does not create a feature pooling layer, similar to the average and max pooling layers .

Dropout was not included in the function set as batch normalization has been shown to perform a similar role of regularization of a network as dropout does [14]. Including only batch normalization in the function set still means that there is a way to regularize networks while reducing the search space due to the reduced function set. Dropout was chosen to be removed due to the negative effect that it can have on the rate of convergence and higher overall accuracy in networks with high levels of dropout compared with networks with low or no dropout. This exaggerates the problem where the best networks, after training for 10 epochs, are not the same as the best networks after training for 60 epochs. The effect of batch normalization on convergence has not been well studied.

6.4 Results and Discussions

Table 6.1: Accuracy on Test set

	Accuracy at 10 epochs	Accuracy at 60 epochs	Accuracy of voting
Run 1	72.3%	72.2%	77.0%
Run 2	68.2%	69.9%	72.3%
Run 3	73.4%	75.7%	80.8%
Run 4	68.3%	71.9%	74.4%
Run 5	67.5%	73.8%	86.0%

The Pearson's correlation between the diversity of the individuals in the voting pool and increase in accuracy from using voting as opposed to the fittest individual was 0.63. This correlation represents a weak correlation between the differences in structure in the tree and diversity in the instances in the test set that were classified incorrectly by the neural networks. There are two reasons for this. Firstly, the diversity in the tree structures does not necessarily correlate to a large difference in the neural networks. Diversity in the tree structure can arise from the order of children been important. This is not the case with a branch node where the order of the second two children is irrelevant so reversing the order of the can lead to high diversity but not change the neural network produced by the tree. Another way that different trees can produce the same neural network is if there are pooling nodes where the input is too small so a pooling layer is not created. The second reason for the weak correlation is that different structures of the neural networks do not necessarily result in diversity in the output of the network. This may be because in some instances the test set might require more powerful networks to be able to learn how to extract the class of the object and less than half of the networks in the voting pool were able to classify these instances.

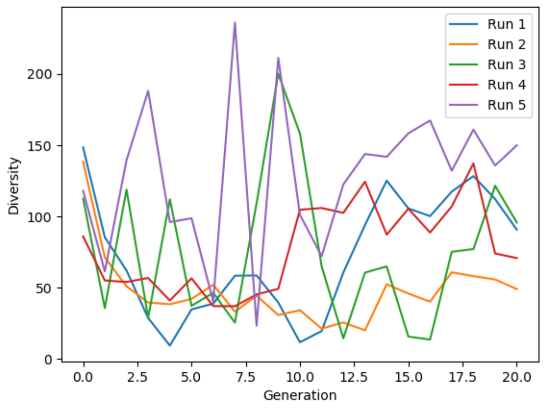


Figure 6.4: Diversity of 5 Fittest individuals

The diversity of the building block was much lower initially than the other approaches. This was expected as the initial population is built from a limited set of structures. However, the diversity did not increase significantly after the initial population due to mutation bringing in new structures.

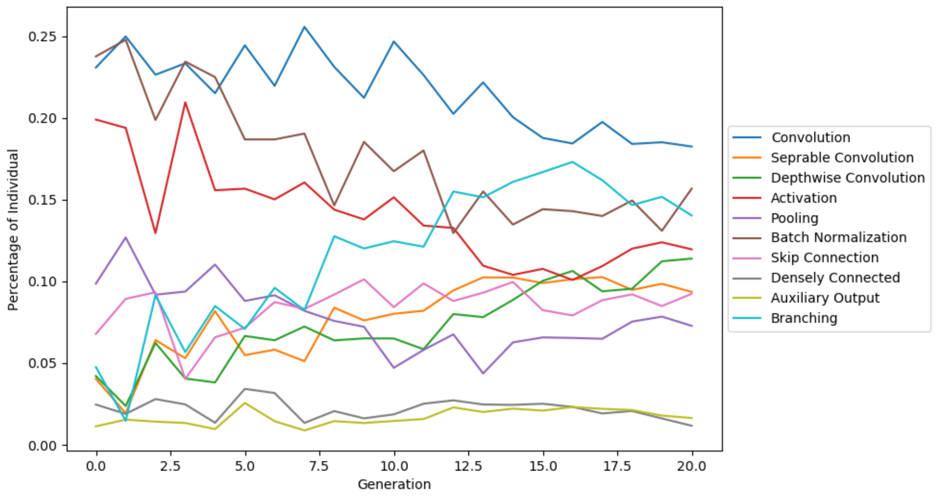


Figure 6.5: Composition of 5 fittest individuals each generation

There were large changes in the composition of the best individuals between the first generation and the last generation. This was somewhat expected as half the initial population was generated using the building blocks which had different proportions of nodes compared with the function set used during training. A good example of this is the decrease in the use of convolution nodes and corresponding increase in depthwise and separable convolution nodes as only convolution nodes were used in the building blocks. One unexpected aspect was the increase in the use of branching nodes which was greater than the other algorithms.

Table 6.2: Classification results on the CIFAR-10 dataset. 95% confidence interval was used over 5 runs of each algorithm.

Algorithm	Accuracy on CIFAR-10	Parameters	Seach cost (GPU days)
GPipe (AmoebaNet)	99%	557M	5250
EfficientNet	98.9%	64M	-
ProxylessNAS	97.92%	5.7M	8
DenseNet	96.54%	27.2M	-
Neural Architecture Search	96.4%	7.1M	22400
GP-CNAS	94.57%	9.7M	180
CGP-CNN	94.02%	1.7M	27
GPCNN	81.2± 2.5%	8.9±9.5M	2.0±1.2
GPCNN with Partial Subtree Crossover	82.2±3.4%	4.7±3.5M	1.0±0.9
GPCNN with Variable Fitness Evaluation	80.5±5.8%	9.6±13.2M	2.0±2.3
GPCNN with Building Block Initialization	78.1±6.4%	3.7±5.5M	1.6±2.4

Base GPCNN did not achieve an accuracy close to that of existing algorithms when evaluating them on the CIFAR 10 dataset. No alterations that were made to the base GPCNN had a significant effect on the final accuracy of the algorithm. The main reason was the lack of runs due to constraints in computational power available. The algorithms developed were run for significantly less time than competing algorithms and it was only feasible to do 5 runs for each algorithm. In all the proposed algorithms except variable fitness evaluation, it appeared that they would continue to improve if run with more generations and more individuals in the population.

Table 6.3: Number of unique Individuals in for each algorithm

Algorithm	Unique Individuals
GPCNN	655±15
GPCNN with Partial Subtree Crossover	698±30
GPCNN with Variable Fitness Evaluation	636±26
GPCNN with Building Block Initialization	647±35

The number of individuals evaluated was higher when using the partial subtree crossover than for the other algorithms. When using the partial subtree crossover operator there are more options to exchange which reduces the chance that crossover or mutation managed to reconstruct individuals. Variable fitness evaluation had the least number of individuals evaluated. This will be because of the low diversity so the crossover and mutation would have a higher chance at changing an individual to be the same as an existing individual.

6.5 Chapter Summary

This chapter shows how domain knowledge can be incorporated into the GPCNN algorithm. This was done through the function set that was used at different parts of the algorithm. Even with including domain knowledge, there was no significant difference in accuracy between the first generation from the building block initialization and the random trees generated from the normal function set. Ideas that make hand-crafted networks perform well do not necessarily perform well when combined.

Chapter 7

Conclusions

The work in this project is the first research into the use of tree-based GP to automate the architecture design of CNNs. While the approach did not achieve competitive results with existing hand-crafted CNN architectures or NAS algorithms, it showed that using tree-based GP is a viable approach. The results presented in this report can be used as a baseline for future research in this area.

A number of modifications on the base GP operators were explored. These were a partial subtree crossover operator, variable fitness evaluation and building block initialization. The different approaches did not have a significant effect on the accuracy or search time of the algorithm. However, most approaches did not converge and looked like they would continue to see improvements in accuracy if training was continued.

- Partial subtree crossover looked like it had promise however it was much more destructive on average compared to the traditional crossover. This hid any benefits that exchanging partial subtree had.
- Variable fitness evaluation put significant pressure on the algorithm to converge and caused GPCNN to converge much faster than the other proposed modifications so while the final accuracy was similar there does not seem to be much room to improve with extra training, unlike the other algorithms.
- Building block initialization caused low diversity compared to the other algorithms which lead to it training slower.

The use of genetic programming led to a large search space which is expensive to explore. Using limited hardware such as only one GPU per run means that the algorithm takes significant time to converge or for results similar to the state-of-the-art methods to be produced. This is due to organisations such as Google that have a large amount of resources at their disposal and can therefore run their algorithms for a long time.

7.1 Future Work

As the learning capacity of GPCNN has not been fully explored in this project due to computational restrictions, it would be valuable to run the base GPCNN algorithm and the modifications until they converge. The learning capacity of each would then be established and can be compared. Then the effect of partial subtree crossover will be able to be evaluated with respect to the learning capacity and, furthermore, potentially validate the expectation that it would increase the learning capacity by making exploitation easier.

Using data set augmentation while doing the final training of the individuals would be helpful in allowing the top individuals to train to higher levels of accuracy by not over-training as quickly and becoming more generalized . Both [40, 42, 59] use the same dataset augmentation on the CIFAR-10 dataset. The augmentation that they used is by padding the images with 4 zeros and taking a random 32x32 crop as well as a random horizontal flip.

7.1.1 Partial Subtree Crossover

Partial subtree crossover could be improved by reducing the average size of the partial sub-trees that are exchanged and so reduce how destructive the operator is. A proposed method could be to choose a random node as the root of the partial subtree then choose a random subtree. This would ensure that the partial subtree crossover is less destructive than the sub-tree crossover as the maximum size of the partial subtrees would be smaller than or equal to the size of the complete subtree.

A similar approach could be used with mutation as well as crossover. This would select a random partial subtree in an individual and replace it with a randomly generated partial subtree. This would require randomly generating partial subtree that matches a signature, further increasing the benefits of the partial subtree crossover.

7.1.2 Building Block Initialization

The initial building blocks needed to have more diversity in the initial population. It would also be interesting to seed the population with a few handcrafted individuals rather than only using randomly generated individuals.

Another approach to increasing the diversity at initialization would be to use the build-ing blocks in conjunction with the function set to build individuals. The effective depth of the building blocks would need to be tracked so that during initializing the tree can be kept within the depth limit when the building blocks get expanded.

Bibliography

[1] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," *arXiv preprint arXiv:1802.01548*, 2018.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[3] J. R. Sampson, "Adaptation in natural and artificial systems (john h. holland)," 1976.

[4] J. F. Peters, *Foundations of Computer Vision: Computational Geometry, Visual Image Structures and Object Shape Detection*, vol. 124. Springer, 2017.

[5] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[6] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[7] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *arXiv preprint arXiv:1811.06965*, 2018.

[8] H. Touvron, A. Vedaldi, M. Douze, and H. Jégou, "Fixing the train-test resolution discrepancy," *arXiv preprint arXiv:1906.06423*, 2019.

[9] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *arXiv preprint arXiv:1905.11946*, 2019.

[10] A. Shah, E. Kadam, H. Shah, S. Shinde, and S. Shingade, "Deep residual networks with exponential linear unit," *arXiv preprint arXiv:1604.04112*, 2016.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *European conference on computer vision*, pp. 630–645, Springer, 2016.

[12] D. Han, J. Kim, and J. Kim, "Deep pyramidal residual networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5927–5935, 2017.

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

[14] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[15] R. H. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung, "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit," *Nature*, vol. 405, no. 6789, p. 947, 2000.

[16] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.

[17] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[18] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, “Inception-v4, inception-resnet and the impact of residual connections on learning,” in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[19] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

[20] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[21] G. Larsson, M. Maire, and G. Shakhnarovich, “Fractalnet: Ultra-deep neural networks without residuals,” *arXiv preprint arXiv:1605.07648*, 2016.

[22] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, “Aggregated residual transformations for deep neural networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1492–1500, 2017.

[23] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning internal representations by error propagation,” tech. rep., California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[24] J. R. Koza and J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*, vol. 1. MIT press, 1992.

[25] H. Cai, L. Zhu, and S. Han, “Proxylessnas: Direct neural architecture search on target task and hardware,” *arXiv preprint arXiv:1812.00332*, 2018.

[26] S. Luke and L. Spector, “A revised comparison of crossover and mutation in genetic programming,” *Genetic Programming*, vol. 98, no. 208-213, p. 55, 1998.

[27] W. Banzhaf and W. B. Langdon, “Some considerations on the reason for bloat,” *Genetic Programming and Evolvable Machines*, vol. 3, no. 1, pp. 81–91, 2002.

[28] S. Bleuler, M. Brack, L. Thiele, and E. Zitzler, “Multiobjective genetic programming: Reducing bloat using spea2,” in *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No. 01TH8546)*, vol. 1, pp. 536–543, IEEE, 2001.

[29] J. Sadeghi, S. Sadeghi, and S. T. A. Niaki, “Optimizing a hybrid vendor-managed inventory and transportation problem with fuzzy demand: an improved particle swarm optimization algorithm,” *Information Sciences*, vol. 272, pp. 126–144, 2014.

[30] J. Kennedy and R. Eberhart, “Particle swarm optimization,” in *Proceedings of ICNN’95 - International Conference on Neural Networks*, vol. 4, pp. 1942–1948 vol.4, Nov 1995.

[31] D. J. Montana, “Strongly typed genetic programming,” *Evolutionary computation*, vol. 3, no. 2, pp. 199–230, 1995.

[32] H. Al-Sahaf, A. Song, K. Neshatian, and M. Zhang, "Two-tier genetic programming: Towards raw pixel-based image classification," *Expert Systems with Applications*, vol. 39, no. 16, pp. 12291–12301, 2012.

[33] B. Evans, "Genetic programming for evolutionary deep learning for image classification," 2017.

[34] Y. Bi, B. Xue, and M. Zhang, "An automatic feature extraction approach to image classification using genetic programming," in *International Conference on the Applications of Evolutionary Computation*, pp. 421–438, Springer, 2018.

[35] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[36] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," tech. rep., Citeseer, 2009.

[37] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," *arXiv preprint arXiv:1812.00332*, 2018.

[38] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

[39] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, pp. 2951–2959, 2012.

[40] M. Suganuma, S. Shirakawa, and T. Nagao, "A genetic programming approach to designing convolutional neural network architectures," in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 497–504, ACM, 2017.

[41] Y. Zhu, Y. Yao, Z. Wu, Y. Chen, G. Li, H. Hu, and Y. Xu, "Gp-cnas: Convolutional neural network architecture search with genetic programming," *arXiv preprint arXiv:1812.07611*, 2018.

[42] Y. Sun, B. Xue, M. Zhang, and G. G. Yen, "Automatically designing cnn architectures using genetic algorithm for image classification," *arXiv preprint arXiv:1808.03818*, 2018.

[43] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," *Statistics and computing*, vol. 4, no. 2, pp. 87–112, 1994.

[44] Y. Sun, B. Xue, M. Zhang, and G. G. Yen, "Evolving deep convolutional neural networks for image classification," *IEEE Transactions on Evolutionary Computation*, 2019.

[45] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.

[46] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, "Hierarchical representations for efficient architecture search," *arXiv preprint arXiv:1711.00436*, 2017.

[47] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, *et al.*, "Evolving deep neural networks," in *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pp. 293–312, Elsevier, 2019.

[48] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

[49] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural networks: Tricks of the trade*, pp. 9–48, Springer, 2012.

[50] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2921–2929, 2016.

[51] L. Bottou, "Stochastic gradient descent tricks," in *Neural networks: Tricks of the trade*, pp. 421–436, Springer, 2012.

[52] L. Prechelt, "Early stopping-but when?," in *Neural Networks: Tricks of the trade*, pp. 55–69, Springer, 1998.

[53] A. Ekárt and S. Z. Németh, "A metric for genetic programs and fitness sharing," in *European Conference on Genetic Programming*, pp. 259–270, Springer, 2000.

[54] S. Rahnamayan, H. R. Tizhoosh, and M. M. Salama, "A novel population initialization method for accelerating evolutionary algorithms," *Computers & Mathematics with Applications*, vol. 53, no. 10, pp. 1605–1614, 2007.

[55] P. A. Diaz-Gomez and D. F. Hougen, "Initial population for genetic algorithms: A metric approach.," in *Gem*, pp. 43–49, 2007.

[56] K. Sastry, "Efficient cluster optimization using extended compact genetic algorithm with seeded population," in *In Proceedings of the Optimization by Building and Using Probabilistic Models OBUPM Workshop at the Genetic and Evolutionary Computation Conference (GECCO-2001 OBUPM*, Citeseer, 2001.

[57] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[58] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1251–1258, 2017.

[59] W. Irwin-Harris, Y. Sun, B. Xue, and M. Zhang, "A graph-based encoding for evolutionary convolutional neural network architecture design," in *2019 IEEE Congress on Evolutionary Computation (CEC)*, pp. 546–553, IEEE, 2019.

Appendices

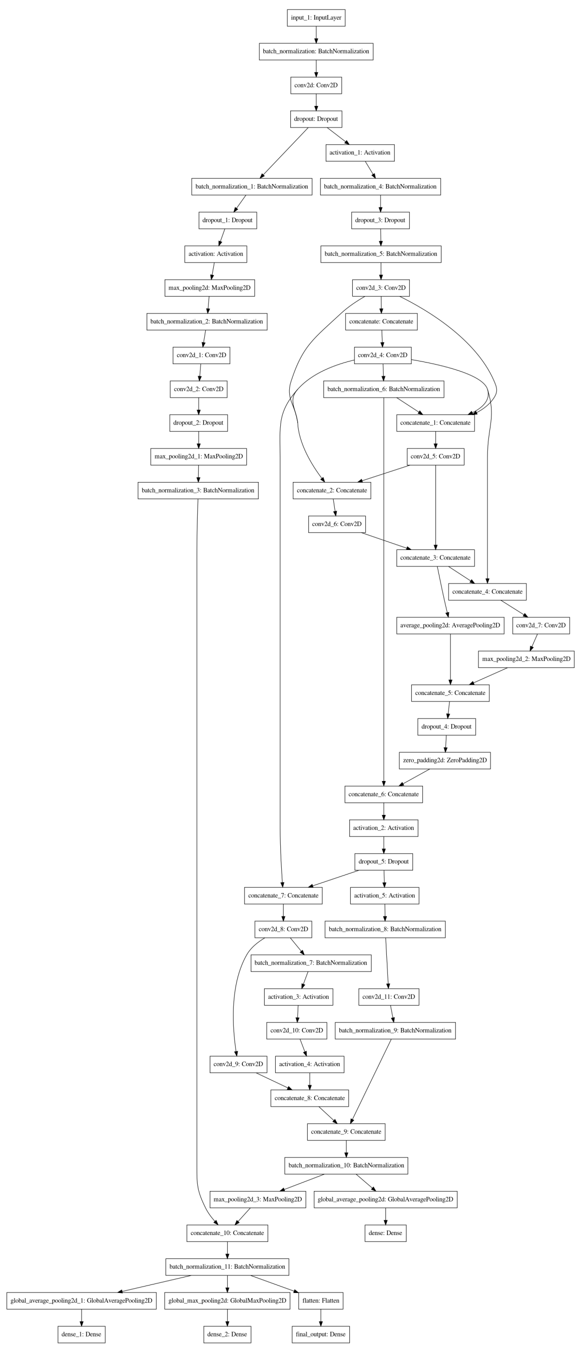

input_1: InputLayer
batch_normalization: BatchNormalization
conv2d: Conv2D
dropout: Dropout
activation_1: Activation
batch_normalization_1: BatchNormalization
batch_normalization_4: BatchNormalization
dropout_1: Dropout
dropout_3: Dropout
activation: Activation
batch_normalization_5: BatchNormalization
max_pooling2d: MaxPooling2D
conv2d_3: Conv2D
batch_normalization_2: BatchNormalization
concatenate: Concatenate
conv2d_1: Conv2D
conv2d_4: Conv2D
conv2d_2: Conv2D
batch_normalization_6: BatchNormalization
dropout_2: Dropout
concatenate_1: Concatenate
max_pooling2d_1: MaxPooling2D
conv2d_5: Conv2D
batch_normalization_3: BatchNormalization
concatenate_2: Concatenate
conv2d_6: Conv2D
concatenate_3: Concatenate
concatenate_4: Concatenate
average_pooling2d: AveragePooling2D
conv2d_7: Conv2D
max_pooling2d_2: MaxPooling2D
concatenate_5: Concatenate
dropout_4: Dropout
zero_padding2d: ZeroPadding2D
concatenate_6: Concatenate
activation_2: Activation
dropout_5: Dropout
concatenate_7: Concatenate
activation_5: Activation
conv2d_8: Conv2D
batch_normalization_8: BatchNormalization
batch_normalization_7: BatchNormalization
activation_3: Activation
conv2d_11: Conv2D
conv2d_10: Conv2D
batch_normalization_9: BatchNormalization
conv2d_9: Conv2D
activation_4: Activation
concatenate_8: Concatenate
concatenate_9: Concatenate
batch_normalization_10: BatchNormalization
max_pooling2d_3: MaxPooling2D
global_average_pooling2d: GlobalAveragePooling2D
concatenate_10: Concatenate
dense: Dense
batch_normalization_11: BatchNormalization
global_average_pooling2d_1: GlobalAveragePooling2D
global_max_pooling2d: GlobalMaxPooling2D
flatten: Flatten
dense_1: Dense
dense_2: Dense
final_output: Dense

Figure 1: CNN architecture from best individual from base GPCNN

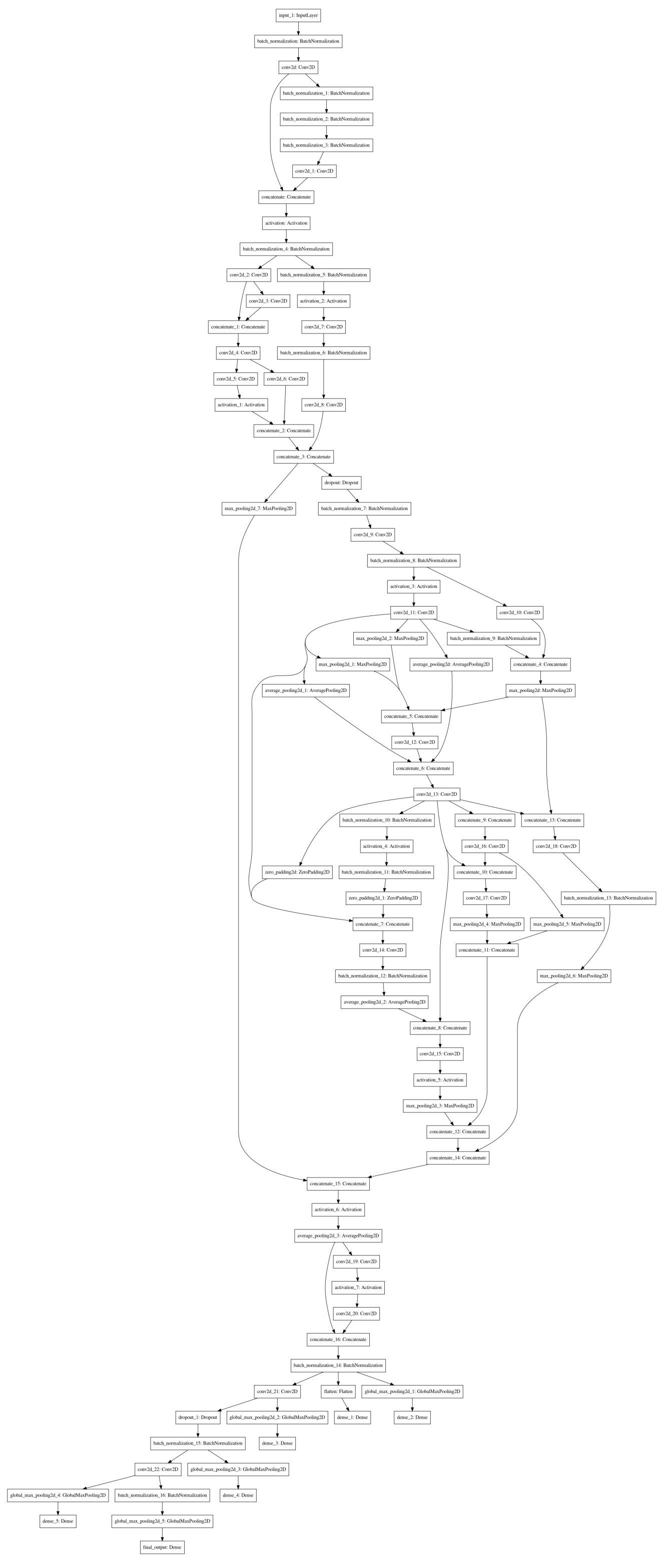
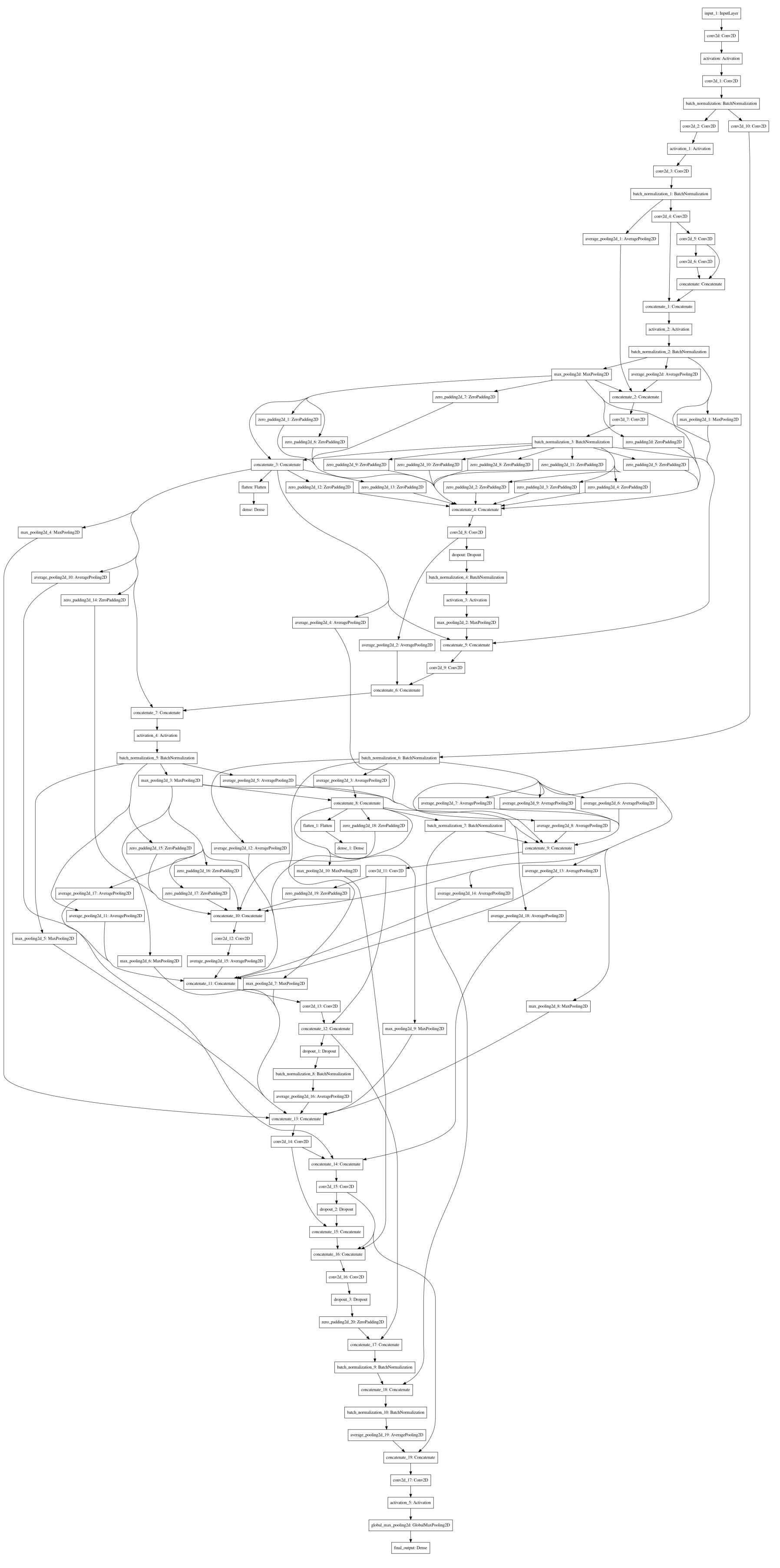

input_1: InputLayer
batch_normalization: BatchNormalization
conv2d: Conv2D
batch_normalization_1: BatchNormalization
batch_normalization_2: BatchNormalization
batch_normalization_3: BatchNormalization
conv2d_1: Conv2D
concatenate: Concatenate
activation: Activation
batch_normalization_4: BatchNormalization
conv2d_2: Conv2D
batch_normalization_5: BatchNormalization
conv2d_3: Conv2D
activation_2: Activation
concatenate_1: Concatenate
conv2d_7: Conv2D
conv2d_4: Conv2D
batch_normalization_6: BatchNormalization
conv2d_5: Conv2D
conv2d_6: Conv2D
activation_1: Activation
conv2d_8: Conv2D
concatenate_2: Concatenate
concatenate_3: Concatenate
dropout: Dropout
max_pooling2d_7: MaxPooling2D
batch_normalization_7: BatchNormalization
conv2d_9: Conv2D
batch_normalization_8: BatchNormalization
activation_3: Activation
conv2d_11: Conv2D
conv2d_10: Conv2D
max_pooling2d_2: MaxPooling2D
batch_normalization_9: BatchNormalization
max_pooling2d_1: MaxPooling2D
average_pooling2d: AveragePooling2D
concatenate_4: Concatenate
average_pooling2d_1: AveragePooling2D
max_pooling2d: MaxPooling2D
concatenate_5: Concatenate
conv2d_12: Conv2D
concatenate_6: Concatenate
conv2d_13: Conv2D
batch_normalization_10: BatchNormalization
concatenate_9: Concatenate
concatenate_13: Concatenate
activation_4: Activation
conv2d_16: Conv2D
conv2d_18: Conv2D
zero_padding2d: ZeroPadding2D
batch_normalization_11: BatchNormalization
concatenate_10: Concatenate
zero_padding2d_1: ZeroPadding2D
conv2d_17: Conv2D
batch_normalization_13: BatchNormalization
concatenate_7: Concatenate
max_pooling2d_4: MaxPooling2D
max_pooling2d_5: MaxPooling2D
conv2d_14: Conv2D
concatenate_11: Concatenate
batch_normalization_12: BatchNormalization
max_pooling2d_6: MaxPooling2D
average_pooling2d_2: AveragePooling2D
concatenate_8: Concatenate
conv2d_15: Conv2D
activation_5: Activation
max_pooling2d_3: MaxPooling2D
concatenate_12: Concatenate
concatenate_14: Concatenate
concatenate_15: Concatenate
activation_6: Activation
average_pooling2d_3: AveragePooling2D
conv2d_19: Conv2D
activation_7: Activation
conv2d_20: Conv2D
concatenate_16: Concatenate
batch_normalization_14: BatchNormalization
conv2d_21: Conv2D
flatten: Flatten
global_max_pooling2d_1: GlobalMaxPooling2D
dropout_1: Dropout
global_max_pooling2d_2: GlobalMaxPooling2D
dense_1: Dense
dense_2: Dense
batch_normalization_15: BatchNormalization
dense_3: Dense
conv2d_22: Conv2D
global_max_pooling2d_3: GlobalMaxPooling2D
global_max_pooling2d_4: GlobalMaxPooling2D
batch_normalization_16: BatchNormalization
dense_4: Dense
dense_5: Dense
global_max_pooling2d_5: GlobalMaxPooling2D
final_output: Dense

Figure 2: CNN architecture from best individual from GPCNN with Parital Subtree crossover



input_1: InputLayer
conv2d: Conv2D
activation: Activation
conv2d_1: Conv2D
batch_normalization: BatchNormalization
conv2d_2: Conv2D
conv2d_10: Conv2D
activation_1: Activation
conv2d_3: Conv2D
batch_normalization_1: BatchNormalization
conv2d_4: Conv2D
average_pooling2d_1: AveragePooling2D
conv2d_5: Conv2D
conv2d_6: Conv2D
concatenate: Concatenate
concatenate_1: Concatenate
activation_2: Activation
batch_normalization_2: BatchNormalization
max_pooling2d: MaxPooling2D
average_pooling2d: AveragePooling2D
zero_padding2d_7: ZeroPadding2D
concatenate_2: Concatenate
zero_padding2d_1: ZeroPadding2D
max_pooling2d_1: MaxPooling2D
conv2d_7: Conv2D
zero_padding2d_6: ZeroPadding2D
batch_normalization_3: BatchNormalization
zero_padding2d: ZeroPadding2D
concatenate_3: Concatenate
zero_padding2d_9: ZeroPadding2D
zero_padding2d_10: ZeroPadding2D
zero_padding2d_8: ZeroPadding2D
zero_padding2d_11: ZeroPadding2D
zero_padding2d_5: ZeroPadding2D
flatten: Flatten
zero_padding2d_12: ZeroPadding2D
zero_padding2d_13: ZeroPadding2D
zero_padding2d_2: ZeroPadding2D
zero_padding2d_3: ZeroPadding2D
zero_padding2d_4: ZeroPadding2D
dense: Dense
concatenate_4: Concatenate
max_pooling2d_4: MaxPooling2D
conv2d_8: Conv2D
dropout: Dropout
average_pooling2d_10: AveragePooling2D
batch_normalization_4: BatchNormalization
zero_padding2d_14: ZeroPadding2D
activation_3: Activation
average_pooling2d_4: AveragePooling2D
max_pooling2d_2: MaxPooling2D
average_pooling2d_2: AveragePooling2D
concatenate_5: Concatenate
conv2d_9: Conv2D
concatenate_6: Concatenate
concatenate_7: Concatenate
activation_4: Activation
batch_normalization_5: BatchNormalization
batch_normalization_6: BatchNormalization
max_pooling2d_3: MaxPooling2D
average_pooling2d_5: AveragePooling2D
average_pooling2d_3: AveragePooling2D
average_pooling2d_7: AveragePooling2D
average_pooling2d_9: AveragePooling2D
average_pooling2d_6: AveragePooling2D
concatenate_8: Concatenate
flatten_1: Flatten
zero_padding2d_16: ZeroPadding2D
batch_normalization_7: BatchNormalization
average_pooling2d_8: AveragePooling2D
dense_1: Dense
zero_padding2d_15: ZeroPadding2D
average_pooling2d_12: AveragePooling2D
concatenate_9: Concatenate
zero_padding2d_16: ZeroPadding2D
max_pooling2d_10: MaxPooling2D
conv2d_11: Conv2D
average_pooling2d_13: AveragePooling2D
average_pooling2d_17: AveragePooling2D
zero_padding2d_17: ZeroPadding2D
zero_padding2d_19: ZeroPadding2D
average_pooling2d_14: AveragePooling2D
average_pooling2d_11: AveragePooling2D
concatenate_10: Concatenate
average_pooling2d_18: AveragePooling2D
max_pooling2d_5: MaxPooling2D
conv2d_12: Conv2D
max_pooling2d_6: MaxPooling2D
average_pooling2d_15: AveragePooling2D
concatenate_11: Concatenate
max_pooling2d_7: MaxPooling2D
conv2d_13: Conv2D
max_pooling2d_8: MaxPooling2D
concatenate_12: Concatenate
max_pooling2d_9: MaxPooling2D
dropout_1: Dropout
batch_normalization_8: BatchNormalization
average_pooling2d_16: AveragePooling2D
concatenate_13: Concatenate
conv2d_14: Conv2D
concatenate_14: Concatenate
conv2d_15: Conv2D
dropout_2: Dropout
concatenate_15: Concatenate
concatenate_16: Concatenate
conv2d_16: Conv2D
dropout_3: Dropout
zero_padding2d_20: ZeroPadding2D
concatenate_17: Concatenate
batch_normalization_9: BatchNormalization
concatenate_18: Concatenate
batch_normalization_10: BatchNormalization
average_pooling2d_19: AveragePooling2D
concatenate_19: Concatenate
conv2d_17: Conv2D
activation_5: Activation
global_max_pooling2d: GlobalMaxPooling2D
final_output: Dense

Figure 3: CNN architecture from best individual from GPCNN with Variable Fitness evaluation

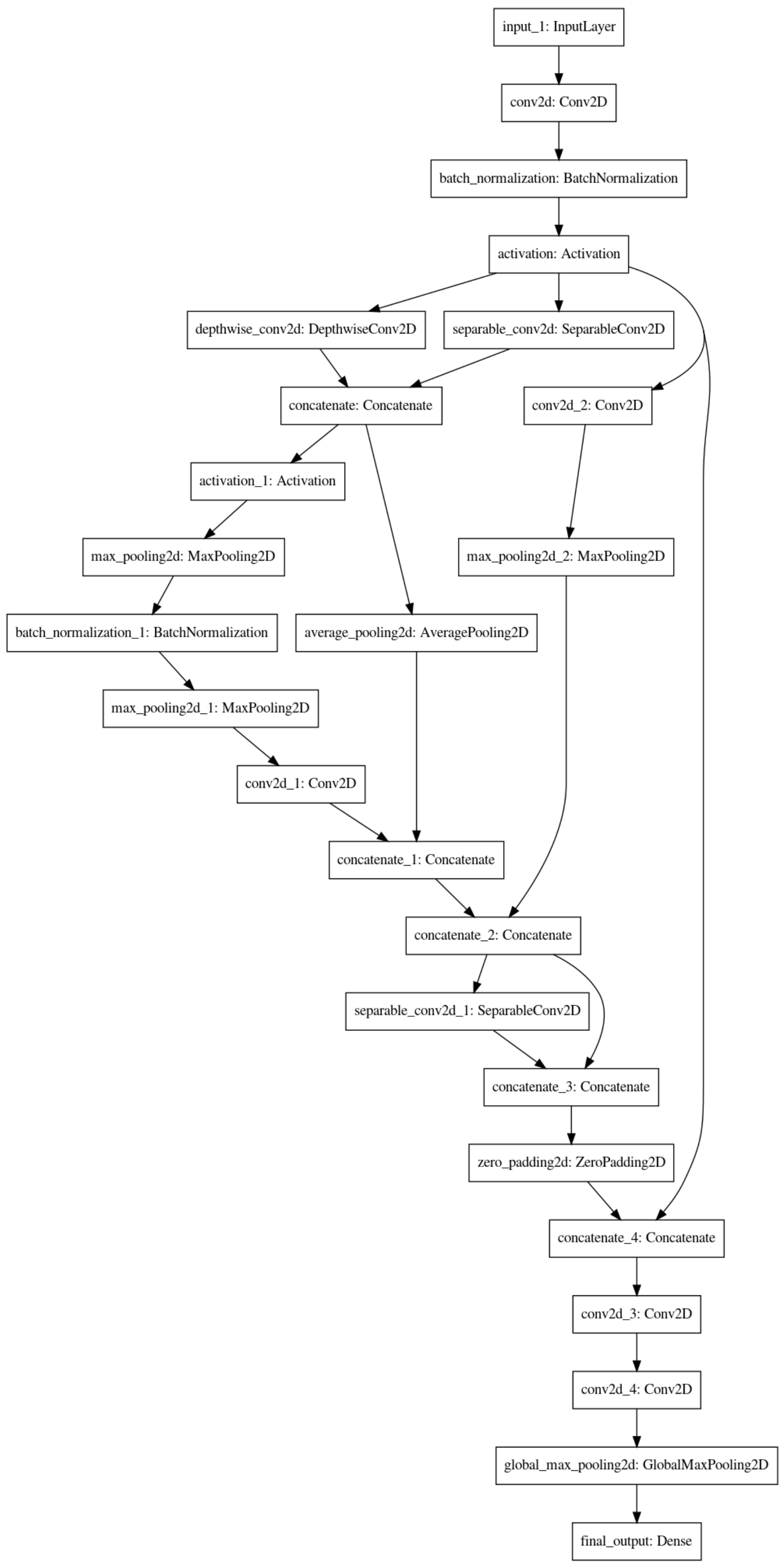

input_1: InputLayer
conv2d: Conv2D
batch_normalization: BatchNormalization
activation: Activation
depthwise_conv2d: DepthwiseConv2D
separable_conv2d: SeparableConv2D
concatenate: Concatenate
conv2d_2: Conv2D
activation_1: Activation
max_pooling2d: MaxPooling2D
max_pooling2d_2: MaxPooling2D
batch_normalization_1: BatchNormalization
average_pooling2d: AveragePooling2D
max_pooling2d_1: MaxPooling2D
conv2d_1: Conv2D
concatenate_1: Concatenate
concatenate_2: Concatenate
separable_conv2d_1: SeparableConv2D
concatenate_3: Concatenate
zero_padding2d: ZeroPadding2D
concatenate_4: Concatenate
conv2d_3: Conv2D
conv2d_4: Conv2D
global_max_pooling2d: GlobalMaxPooling2D
final_output: Dense

Figure 4: CNN architecture from best individual from GPCNN with Building Block Initialization