Improving CoDeepNEAT with Elitism and Stability

Shane Acton
Computer Science
University of Cape Town
Cape Town South Africa
shaneacton745.sa@gmail.com

ABSTRACT

Neuroevolution is an attractive solution to the model hyper-parameters problem, however it generally requires many orders of magnitudes more computational resources than hand crafted models, which is a barrier to independent researchers and small organisations. It is therefore necessary to explore NE solutions which can deliver maximum performance with the minimal required resources. CoDeepNEAT (CDN) is a NeuroEvolution (NE) algorithm which can rapidly evolve novel Deep Neural Network (DNN) architectures. It co-evolves populations of repeating DNN structures along with blueprints which dictate how the structures should be connected. In this paper we offer a set of major extensions to improve the performance of the CDN algorithm without significantly affecting computation time. We propose Elite CDN, named for making CDN truly elitist, something which was missing from CDN previously. Elite CDN introduces a deterministic relationship between CDN's genotype and phenotype (DNN). It demonstrates superior performance when compared to our implementation of CDN in the CIFAR-10[1] dataset. When compared to Reinforcement Learning (RL) based DNN network architecture search, as well as AmoebaNet, which is a successful NE algorithm, we demonstrate that CDN can trade off a moderate loss of performance, for a massive computational time speed-up.

1 INTRODUCTION

In recent years, with the rising availability of computational and data resources, Deep Learning (DL) has quickly overtaken many older Machine Learning (ML) approaches, particularly computer vision and language processing [3,4,5]. As DL itself develops we are seeing larger and more complex models being needed to address the increasingly complex problems which are becoming solvable [5,6,7,8,9,10]. As these models become more complex the number of human selected parameters is increasing. Deciding which regularisation techniques to use and where, as well as the topologies and dimensions used by deep layers can require years of fine tuning, as can be seen with the iterative development of the InceptionNet [5,7,8] Convolutional Neural Network (CNN) from 2014 - 2016. The difficulty in tuning deep learning models can lead to design time eclipsing the training time of any of the given iterations of a network.

AutoML, the process of automating the design of ML solutions, is an attractive solution to the hyper-parameter tuning problem. It trades in human design hours for orders of magnitudes of extra computation, which can be useful for anyone with sufficient resources who is looking to achieve optimal performance of their models with minimal human intervention. Google has begun using domain specific NE algorithms to design State Of The Art (SOTA) models, namely AmoebaNet [11].

CoDeepNEAT (CDN) is a prominent NE approach which evolves DNNs. Its principal contribution is extending the popular ANN evolver NEAT to the domain of Deep Learning, and conceptually promoting the evolution of repeating structures called modules. CDN uses a module system that was inspired by networks such as InceptionNet[5,7,8], which define a few small Artificial Neural Networks (ANN), called modules, and stitch them together with a blueprint. CDN has been shown to create SOTA networks in some domains [1,2]. However, there are some issues with the CDN algorithm, chief among them is the inconsistency of its results. The efficacy of CDN's solutions can vary dramatically from generation to generation [12] due to the fact that the algorithm is not truly elitist. In this paper we will be building an open source implementation[2] of CDN based off of the description of the algorithm by the original authors [1,2]. We will use this CDN implementation to test our proposed extensions, which attempt to address some of these shortcomings.

1.1 Research Question

Our experiments and analysis will be focused on determining whether our extensions, which we describe in section 3, have the potential to meaningfully contribute to the CDN algorithm. CDN has been shown to achieve SOTA results in some problems [1,2], while also being a promising candidate for a computationally cheap NE. We hypothesise our extensions will make CDN an elitist EA, and as such will improve its performance.

2 RELATED WORK

This section is about familiarising the reader with some of the fundamental concepts this paper builds upon. Namely Deep Learning, EA's, Neuroevolution, and finally CDN.

2.1 Deep Learning

An ANN is a weighted directed graph of neurons which is roughly inspired by biological neural networks. The general form of an ANN is a set of input neurons which take in inputs to be processed, a set of output neurons which output an approximation of the target function, and a set of hidden neurons, which are all the neurons which are in between the input and output neurons. ANNs have been shown to be universal function approximators [13].

[1] https://www.cs.toronto.edu/~kriz/cifar.html

[2] https://github.com/sash-a/CoDeepNEAT

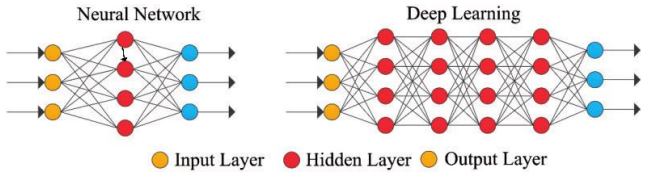


Figure 1: A typical Deep Neural Network (right) compared to a classic ANN (left). Traditional ANNs have fewer structural constraints, allowing inter-layer-connections [14]

Deep learning refers to the creation and training of ANNs with multiple hidden layers. A layer in an ANN is a collection of neurons which are not connected to each other [Figure 1]. Generally layers connect to each other such that the output of one layer gets passed to the input of the next layer in a sequential fashion. However this is not a hard rule, some Deep Neural Network's (DNN) pass outputs of certain layers to many subsequent layers. This is Generally called a skip connection, and was popularised by ResNet [9,10].

A CNN is a deep learning architecture primarily used for computer vision tasks, among other things. In a traditional CNN, images are initially passed through convolutional layers [15], which act as complex feature extractors. A convolutional layer is a limited fully connected layer where neurons are only connected to a subset of the input neurons of the previous layer. This was partly inspired by the neural wiring of a biological visual cortex, and conceptually allows for the idea of spatial locality to be easily included in the CNN. Once the convolutional layers have performed feature extraction, traditionally their outputs would then be passed through fully connected layers, which map the extracted features to the target function. CNNs are currently state-of-the-art in the domain of image recognition [16].

2.2 Neuroevolution

Evolutionary Algorithms (EA) are an optimization technique that use the principles of darwinian evolution to evolve a population of solutions to a problem, towards a defined goal. EAs need a way to find the fittest individuals (solutions) from the population. This is done using a predefined fitness function, a metric which measures how well a solution solves the problem. Convergence towards the optimal solution happens during cycles of reproduction and selection, where the fit survive and pass their properties to later generations via reproduction, thereby sifting out unfit properties [Figure 2]. Reproduction can be asexual, where children are mutated varients of a single parent. It can also be sexual, where children are interpolations of two fit parents. Most EA's represent solutions as genomes. A genome is an encoding of all of a solution's properties, meaning you can use a genome to generate the solution it represents. Solutions then have two forms: genotype and phenotype, where genotype is the genome encoding, and the phenotype is the solution itself. Instances of a phenotype can directly be evaluated against the given problem. These objects may be complex data types. A genotype is an encoding of a phenotype, usually a string or binary sequence. Typically it can be more easily mutated or reproduced than the phenotype form. Sexual reproduction of genomes is often referred to as crossover.

NE is the process of using EA's to design ANNs. In a NE algorithm, the individuals being evolved are ANNs and the fitness function that is maximised is the performance of the ANN in solving a given

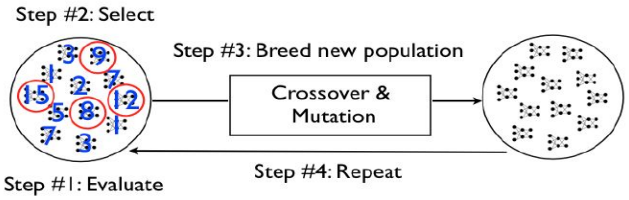


Figure 2: The Neuroevolution cycle. This diagram depicts a population of ANNs which are selected and reproduced to form subsequent population generations [17]

problem. Some NE approaches simultaneously evolve the topologies of the ANNs as well as their weights. Others only evolve the topologies, using more traditional methods to train the weights of the ANNs.

2.3 CoDeepNEAT

The performance of an ANN is highly dependent on its topology. NEAT [18] is a traditional NE method. It attempts to find both a network's optimal topology and weights using evolutionary methods. A difficulty when using EAs to search for better ANN topologies is that crossing over two good but dissimilar solutions may create a damaged or useless offspring. NEAT solves this by introducing historical markings. These markings allow individuals to keep track of their parents and topological features. NEAT will only crossover networks with similar features which minimizes the creation of defective offspring [18] and maximizes its efficiency.

CDN is a recently developed co-evolutionary, multi-objective NE algorithm which has achieved state-of-the-art results [1,2]. CDN is an extension of NEAT, however it is significantly different. First, it does not evolve network weights. Instead CDN updates its network's weights via backpropagation. Second, individuals are co-evolved using a system of blueprints and modules. However, CDN still uses NEATs main contribution, historical markings.

The blueprint and module system is the most significant contribution of CDN [1,2]. Blueprints are graphs where each node points to a specific species of module, each of which is essentially a collection from which that blueprint node will select a module. Modules represent an ANN as a graph, where each node in the module is a deep learning layer in the ANN. Blueprints essentially select modules from their species, and dictate how they are stitched together to form a full DNN. The combination of a blueprint and a set of modules it picks makes up the genotype of CDN.

CDN starts by initialising a population of blueprints and modules. It then repeats a cycle of selection and reproduction typical of an EA. Since CDN supports MOO, ranking is done via pareto-based [19] non-dominated [19,20] sorting of individuals with respect to their scores in each objective. In the case where only one objective is provided, individuals are simply ranked in order of their score, typically accuracy. To evaluate blueprint and module individuals, they must be parsed into functional DNNs. This is done by looping through each blueprint in the blueprint population, and parsing each into multiple separate DNNs. To create one DNN from a blueprint, each of the blueprints nodes is converted from a pointer to a module species - to a single module which is randomly sampled from that species. These modules are then connected to each other according

to the connections between the blueprints nodes. Since each module represents a small DNN, the resulting parsed blueprint, which connects these small DNNs together, represents the full DNN [Figure 3]. This full DNN created from the blueprints connections, and the modules its nodes sampled, is a CDN phenotype.

The phenotype (DNN) is then trained on the training data provided, and tested on testing data. The result of this test provides CDN with its first objective - accuracy. In the default case where the second objective is to minimise network complexity - the DNNs parameters are counted. Using these scores, the solutions are sorted, and the least fit modules and blueprints are removed from the population. The remaining individuals then reproduce to refill the population. A subset of the individuals which are fit enough to reproduce are propogated forward into the next generation, these are the elite blueprints and modules [see section 8.1 table 2 for these ratios].

Evolution of modules and blueprints are related via their co-evolution to form DNNs, however it is noteworthy that they themselves evolve in different ways. Blueprints are collected in a single population where any blueprint is able to reproduce with any other. Modules however are collected in a number of species, where only members of the same species are allowed to reproduce with each other. This is a concept carried forward into CDN from NEAT [18]. The idea proposed is that each species should logically represent a different functional niche. This means that when a blueprint node links to a species, it is choosing to use the niche that the species represents, as when it is parsed into its phenotype DNN - it will randomly sample its module from this species. This linking is done by holding onto the species number[1] of the desired module species. The authors of the CDN paper [1] motivated this co-evolutionary process through the apt observation that modern CNN architectures have tended towards using repeating structures, as is the case in InceptionNet and ResNet [5,7,8,9,10].

When the module population is initialised, the speciation process begins by selecting an arbitrary individual to be the representative of the first species. The representative of a species is used as a benchmark to compare against other individuals via a similarity metric. That is to say the similarity of an individual to the representative decides if it belongs in that species. The similarity metric used is a function of the connections that two graphs don't have in common.

Once the first species has a representative, the remaining individuals in the module population are compared against the representatives of each existing species (only one initially) to decide where they should be placed. If an individual's distance to a species is less than a threshold - called the speciation threshold - it is placed in that species. If the individuals distance to all of the existing species representatives is greater than the speciation threshold - a new species is created, and that individual is made to be its representative. The speciation threshold must be dynamically adjusted throughout the evolutionary run to keep the number of species close to the target [table 2]. The NEAT speciation algorithm specifies that an individual should be placed in the first species which it is found to fit in - not necessarily the species which is fits best in.

The members which lie in a species are defined by that species' representative. This representative must be reselected every

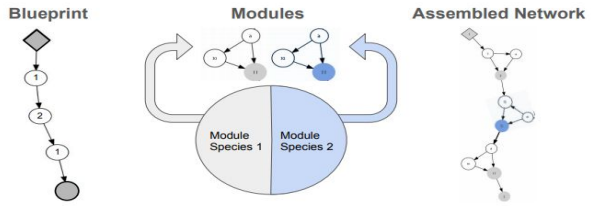


Figure 3: A diagram showing the process of constructing a DNN from a blueprint, by connecting modules sampled from module species [1]

generation. Once a species' members have reproduced - NEAT states that the species' next representative should be picked randomly from the species members.

In CDN the solution being evolved (phenotype) is an assembled DNN made from both blueprints and modules (genotype). The assembly process involves replacing each node in a blueprint with a randomly sampled individual from the species that the blueprint node points to [Figure 3]. Because of this random sampling, each blueprint genome can create many distinct phenotypes. The concept behind this sampling is that each module species should represent a functional niche. Since speciation is based upon a similarity metric, conceptually modules in the same species should be similar to each other. In practice, this process is limited by a naive similarity metric, which only takes into account the number of connections individuals do not share. This means modules within a species may still perform significantly differing functions, which means that the specific module chosen from a species has a large effect on the final DNN.

CDN controls for the random, non-deterministic relationship between genotype (blueprint, module genomes) and phenotype (DNN) by parsing each blueprint into multiple distinct DNNs, and evaluating each separately. Each time the blueprint is parsed it will sample different modules. This way, the modules in a species get explored more fully, and blueprints can somewhat explore their possible phenotypes. However, even in a conservative example case where a blueprint has 3 different module species that its nodes sample from, and each of those species has 10 modules, the space of possible phenotypes which can be created is 10^3. This can be even larger in more extreme cases. As such, the multiple DNNs which are created for evaluation, typically less than 10, is inadequate to fully represent all the possible phenotypes a blueprint can create.

CDN supports elitism [21,22] at the population level, which means the very best blueprints and modules of a generation survive into the next generation in their respective populations. Since the final DNNs that CDN produces are a product of both blueprints and randomly sampled modules, even if the constituent blueprint and modules (genotype) which make up some best DNN *d* in any generation *g* survive into generation *g+1*, it is not guaranteed that the same DNN *d* will be constructed again. Furthermore due to the number of distinct DNNs which could be created by a blueprint, being orders of magnitudes greater than the number of evaluations per blueprint - it is very unlikely that a DNN created in one generation will ever be created again, regardless of how many generations its genotype lives on for. Thus true elitism, where best solutions/phenotypes carry forward, is not a property of CDN.

2.4 Mutation Rates

The mutation rate that an EA uses controls how likely each evolvable parameter is to change each time a solution is mutated. The choice of mutation rate is a very important hyper-parameter for any EA [23]. Furthermore, it has been shown that a well designed dynamic or adaptive mutation rate generally outperforms a static rate which stays constant throughout the run [23]. Adaptive mutation rates change based on live metrics produced by the evolutionary run, a good example of this is Rechenberg's "1/5 success rule" [24], whereby the mutation rate is changed as a function of the proportion of mutations which are successful. Dynamic mutation rates are simpler, they change as a function of time, or generation. Dynamic mutation rates generally decrease over time to allow for rapid exploration of the search space early, followed by fine tuned exploration as the solutions get closer to optimal [23].

3 METHODS

This section details the extensions tested in this paper, as well as The speciation overhaul which is incorporated in the Elite CDN algorithm.

3.1 Elitism improvements

True elitism [21,22] in an evolutionary algorithm requires that the best solutions from a given generation are propagated forward into the next generation. This means that at worst a subsequent generation will have the same maximum performance as the previous, meaning an evolutionary algorithm can guarantee that the maximum performance of a given generation will only increase over time, until convergence.

CDN employs elitism at the module and blueprint population levels. However, because of the random module sampling during construction of the phenotype this is not sufficient to allow for true solution elitism. Even if the modules and blueprint which comprise the best DNN found so far survive, it may take thousands of blueprint to DNN parsings to create it again. Since it has been shown in a number of papers that elitism is hugely beneficial to evolutionary algorithms [21,22], it is therefore worthwhile to explore variations of CDN which allow for more true elitism. Experimentally, the fact that CDN struggles to propagate good solutions forward can be seen in the results of [12] where the maximally performing solution varies hugely from generation to generation, with very good solutions often found early on, but quickly lost. The magnitude of performance drops which can be observed at regular intervals further motivates the need for an improved elitism based CDN.

3.1.1 Module Retention

In classic CDN [1,2], during a parsing from blueprint to DNN, a map of species number to module is created. This is because CDN states that if one blueprint has multiple nodes which point to the same species of module, then upon parsing, each of those nodes must sample the same module. When the same blueprint is parsed again however, those blueprint nodes may pick another module, as long as in that single parsing, there is a unique mapping from species number to module. In classical CDN, this species to module mapping is lost after every parse of the blueprint.

Module retention is about allowing blueprint genomes to hold onto their species to module mappings through generations, reproduction and parsing. Each parsing of a blueprint to a DNN naturally creates its own, often unique, species to module mapping M'_i but only one can be committed into the blueprint genome to be used in later generations. The module retention extension gives blueprints the mapping M'_i from the best performing DNN created by the blueprint (out of the multiple parsings, which created multiple DNNs). This way blueprints are able to explore their phenotype space (the possible DNNs which could be created), and evolve towards using the best possible modules.

Once all parses of a blueprint b have been evaluated and the best M'_i has been given to b, blueprint b will have its own genome mapping GM which we call a full map, meaning the mapping dictates every module which will be used by the blueprint. Once the full evaluation phase has completed, CDN will enter its reproductive step, at which point many of the modules which existed in the previous generation, and were used in genome mappings will die, meaning they will be removed from their population, and will not propagate to the next generation. When modules die, all map entries which pointed to that module must be lost. As a consequence of this, many blueprints will go from being fully mapped to partially mapped, meaning some of their modules will be picked by their genome mapping GM, and some will be sampled as is usual in CDN [1,2].

More formally any blueprint genome has a set of species numbers S which its nodes use to sample modules from. This is simply the union of all of the species numbers used by all of the nodes in the blueprint. Module retention attaches to blueprint genomes, a mapping set GM which maps a subset of the used species $s \subseteq S$ to a set m of modules to be used such that $GM : \{s \rightarrow m\}$. During parsing from blueprint to DNN, a blueprint will first look in its genomes module mapping GM to decide which module to use given a species number. If no mapping exists for that species number in the genome, then the blueprint will sample a module as usual.

Each time the blueprint is parsed, a full mapping set $M'_i = \{s' \rightarrow m'\}$ is created, where s' = S. This full set is created by combining the blueprints genome module mapping GM with the newly created sampling mapping SM_i . The sampling mapping is made by recording which modules were sampled from each species that was not already mapped by the genome. Since each parsing of a blueprint creates a unique SM_i, each parsed DNN has its own full mapping M'_i. After each of the DNNs are evaluated, the most accurate passes its mapping M'_i to the blueprint genome which created it. This way a blueprint has multiple attempts per generation to find a good mapping.

To summarise, a blueprint starts with an empty genome mapping GM. When evaluated for the first time, a number of full mappings M'_i are created from sampling alone, the best of which is given to GM. During reproduction many of the modules mapped by GM die, leaving GM a partial mapping. When the blueprint is evaluated again new full mappings M'_i are created by combining GM and the samplings SM_i. This allows blueprints to hold onto specific modules as long as they survive, and only resample when those held modules die.

A caveat of Module Retention is the question of what to do with species to module maps where the mapped module crosses over into a new species during speciation in the reproductive step. The possible options were to lose the mapping entry; change the blueprint nodes using that mapping to link to the new species number; or keep the blueprint nodes' species the same but use an override such that the module is kept by the nodes mapping to it, despite the fact that it no longer resides in their species. We chose the latter option as we thought it would have the smallest impact on performance. In this case, a blueprint node can continue to map to a module as long as the module lives on, regardless of which species it moves to.

3.1.2 Max Fitness Aggregation

There are a number of conceptual benefits to module retention, one such benefit being its ability to allow blueprints to evolve to use specific modules, while still allowing classic CDN style samplings and repeating structures. Its main function however is to take a step towards making CDN a truly elite algorithm. It does this by making the relationship between genotype and phenotype more deterministic. Meaning if the genotype survives, the phenotype will survive too, which was not guaranteed in classic CDN [1,2].

Conceptually, to achieve true elitism, one would expect that the best DNN of a given generation would be created again in the following generation. Module retention goes some of the way to allow this. A barrier to true elitism not addressed by module retention is the fact that all fitnesses for modules and blueprints are averaged over all of their uses every generation. Since a module which is a part of the best DNN of a generation may also be a part of other bad DNNs, the high fitness it attained in the best DNN will be diluted by all of its subpar uses. This means that the modules used by the best DNN may still come to die in the reproductive step forcing the blueprint to resample modules, meaning the best DNN created is lost.

A similar issue arises with blueprints. If one of the sampling maps SM_i used is excellent, and leads to an effective DNN to be created from its blueprint, the other sampling maps may not be so good, and since these bad sampling maps will be ignored by module retention anyways (since only the best maps get committed to blueprint genomes) - there is no reason to penalise the blueprint for these lacking maps by averaging the evaluations scores together.

The solution to this problem is to replace mean aggregation with max aggregation of blueprints and module fitness. This simple change means a blueprint will be scored on the best mapping it used in evaluation (the mapping which now lies in its genome). And modules used by the best blueprints will survive for that blueprint to use again later. Consider a locally optimal DNN *d*, made by blueprint *b* and a set of modules *M*. Under max fitness aggregation, *b* and all modules in *M* will be assigned the same locally maximal score. Since *b* will be the highest scorer out of all blueprints - and CDN implements population level elitism - *b* is guaranteed to survive. The modules in *M* will all tie for the best performing modules. This is an issue since we would like to ensure that they all survive into the next generation, however the number of elite individuals to keep in the module population may be lower than the size of *M*. The solution is to add a caveat to population based elitism whereby if the number of ties for first place exceeds the number of elite to keep - all of the ties are kept alive. With this final piece of logic, we can guarantee that the best performing DNN of a given generation will ensure the survival of its blueprint and all of the modules it used. With module retention allowing a best blueprint to reuse its modules again in the next generation, we have ensured that the best DNN of any given generation will be recreated in the next generation.

We have named the version of the CDN algorithm with both Module Retention and Max Fitness Aggregation : ModMax CDN

3.1.3 Module Mapping ignores

As the Modmax CDN algorithm gets closer to converging, many blueprints may become fully, or near fully mapped. This is because successful blueprints ensure their own survival, as well as the survival of the modules they use. Fully mapped blueprints are always parsed to the same DNN, since no modules are sampled - only mappings are used. This means that fully mapped blueprints waste their multiple evaluations.

To combat evaluation wastage module-mapping-ignores allow blueprints a chance to ignore one of their mappings and resample from the ignored species. The probability to ignore a mapping is proportional to how fully mapped a blueprint is [see section 8.1 table 2]. A fully mapped blueprint will ignore a mapping entry almost every evaluation, but a blueprint with only a small fraction of mapped species numbers will likely use all of its mappings, not ignoring any. This is because a less mapped blueprint will have enough variation in constructed DNNs due to the random sampling of modules for its unmapped species. To avoid losing elitist properties, the ignores system guarantees that at least one parsing of a blueprint to a DNN will not ignore any part of its mapping. Due to the benefits to exploration provided by mapping ignores, it will be included in the ModMax CDN algorithm by default.

3.3 Mutation Extensions

3.3.1 NEAT Node Breeding

When CDN breeds two parent genomes to create a child, be it a module or blueprint genome, one parent is the better performing parent B, and one is the worse performing parent *W*. *B* and *W* each have sets of nodes N_B and N_W. These nodes are considered the same if they have the same innovation number - which is an idea pioneered by the NEAT paper [18] to ensure meaningful reproduction of graphs. Two nodes considered the same by innovation number don't necessarily have the same attributes eg: convolutional window size. And so when two parents have a node in common, they in fact have two variants of that node. The child created via this reproduction inherits all the nodes in N_B which are not in N_W : $N_{child} = N_B - (N_B \cap N_W)$. For each of the nodes which the two parents have in common - the child inherits the node variant from a random parent.

Under node breeding - instead of inheriting the variant of a common node from a random parent. The child inherits a new node which has the same innovation number - but whose attributes are a natural interpolation of the two parents common node variants. Considering a common node with a historical marking *n*, with two variants : *one-dimensional, fully-connected 100 output-features layer* and

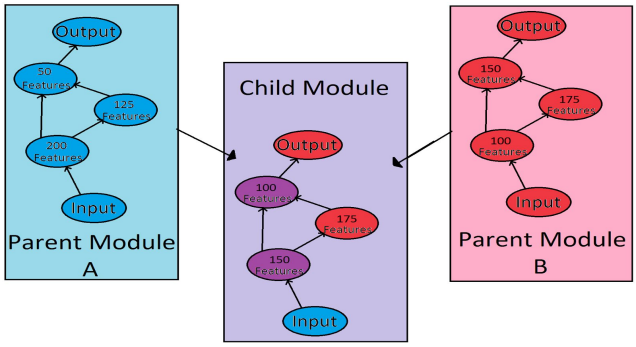


Figure 4: A diagram showing a child being created via crossover. The child inherits some of its nodes(red and blue) directly from its parents. Some nodes (purple) are interpolated via node breeding

one-dimensional, fully-connected 200 output-features layer, standard CDN would give the child either 100 or 200 output features, whereas node breeding would give the child 150 output features. We hypothesise that this would allow the EA to explore the attribute space more quickly and more thoroughly. Node breeding can also be given a probability. A node breed chance of 50% means there is a 50% chance the child will be given an interpolated node, and a 50% chance to receive a copy of the node that either one of its parents have. This way in the aforementioned example the child may inherit an output feature count of either 100, 200 or 150 [Figure 4].

3.3.2 Global Topological and Attribute Mutation Magnitude Adjustment

We propose using two global mutation magnitude/rate modifiers - one for topologies and one for attributes. The progression of these two modifiers should ensure that in early evolution both the topologies and attributes should mutate rapidly. During the intermediate phase topologies should start to stabilize while leaving attributes free to mutate quickly. This is the time that can be used to search the attribute space for more stable topologies. Finally in the end phase of the evolution, both topologies and attributes should stabilise, allowing for fine grained optimisation. The function for these two modifiers can be found in section 8.1 [table 2], and a visualisation can be found in section 8.3 [figure 5].

To create children for the next generation of CDN's populations, parents are selected out of the most fit of the previous generation. A child genome is created by performing crossover with the parents genomes. This child genome is then mutated once, giving all of its attributes a chance to mutate, as well as giving it a chance to change its topology. We propose a concept called mutation magnitude which dictates how much this child genome mutates, and thereby how much the child deviates from its parents. Mutation magnitude is simply a modifier variable added to mutation probabilities as well as the magnitude of attribute changes. A higher mutation magnitude means more properties will change, and they will change by larger amounts. We decided to split mutation magnitude into two distinct functions. First is topological mutation magnitude, which modified all mutation chances which affected the topology of a genome. Second is attribute mutation magnitude, which modifies how likely any given node attribute is to change, as well as how much its value changes by.

Since EAs converge on fitter solutions via selective pressure, we can
say that solutions which are found in later generations should be increasingly close to optimal, assuming the EA is successful. This modification will make mutations more extreme in earlier stages of evolution, and progressively moves towards smaller more stable mutations as CDN closes in on optimal solutions. This is inspired by dynamic mutation rates [23]. We will call this extension GMA-CDN. We hypothesise the GMA-CDN algorithm will rapidly explore the topological and attribute feature space at a high granularity early on, and then make increasingly sensitive changes as algorithm gets closer to convergence [23].

We noted that conceptually - the optimal set of attributes for nodes in a graph depends heavily on the topology of that graph. Consider trying to evolve the optimal set of properties for a graph which is constantly changing shape. This could not be done as the optimal properties would be a moving target. To truly optimise attributes, the topology in which those attributes are applied to must be made to be stable. This was the motivation behind changing the topology and attribute mutation magnitudes separately.

3.3 Speciation Overhaul

We propose two small changes to the speciation process. First, place individuals in the species where they fit best, with respect to their similarity to the species' representative. This is in contrast to the first species they fit in as in NEAT [18] and classic CDN [1,2]. This is done by calculating the similarity of the candidate individual to every existing species' representative, and placing the individual in the species whose representative is most similar according to the similarity metric being used [see section 2.3]. If the distance value (according to the similarity metric) to the most similar representative is above the speciation threshold, then a new species is created, unless the target number of species has already been reached, in which case the speciation threshold is increased.

Secondly, instead of selecting an arbitrary representative for a species each generation - select the member for whom the sum of distances (according to the similarity metric) to all other members in that species is a minimum. We hypothesise that this member is the closest to the centroid of the members in the abstract topological-feature-space that the similarity metric defines. In other words we hypothesise that his member is the most representative member of the species. This process is done for every species individually, each generation after the species perform their reproductive step.

These changes are aimed at increasing the stability in the number of species over the generations, and to make species representatives more meaningful, thereby making species represent more disjoint functionalities with respect to one another.

We have named the version of the CDN algorithm with Module Retention, Max Fitness Aggregation as well as the Speciation Overhaul: Elite CDN

3.4 Feature Multiplier

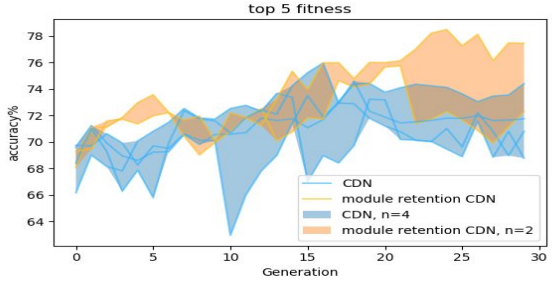
Due to computational and time constraints, it is necessary to evolve DNNs using only a fraction of the training epochs one would normally use to fully train a DNN. Because of this the evolutionary process positively biases DNNs which train quickly. DNNs which have more trainable features/weights typically take more epochs to train to convergence. Thus the evolution process biases against large networks.

We observed that when taking the best graphs evolution could produce and training them to convergence, the results were often not much better than what those graphs achieved in the evolutionary run [table 1], typically around a 3% gain. To alleviate this we propose a feature multiplier which proportionally increases the number of out features/size of each layer in a final DNN created by evolution - essentially creating larger variants of the evolved networks which are able to make use of more training epochs. In this way we hope to continue to experience the computational time benefits of evolving networks with few epochs during evaluation, and still end up with large highly performant networks.

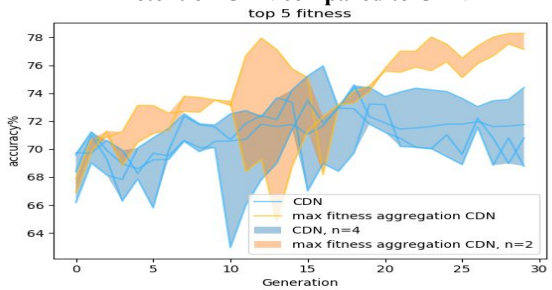
4 RESULTS AND DISCUSSION

4.1 Data

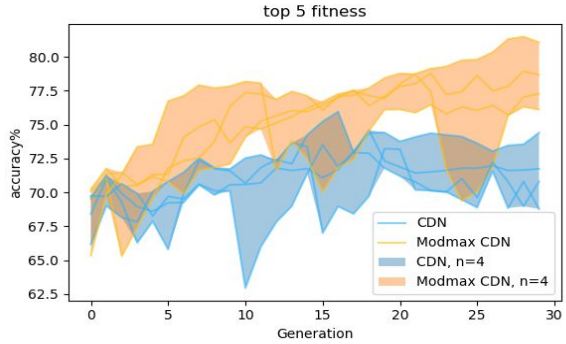
Graphs [1-10] each depict the average of the top 5 DNN evaluation accuracies per generation. Top 5 average was chosen instead of the max accuracy to remove some of the noise which is introduced by training on only 5 epochs. Each line represents a separate evolutionary run. The coloured areas show the band between the max and min performance of each run type. Each band is labeled with an *n* value, this indicates how many evolutionary runs make up the band. See [table 2 and 3] in section 8.1 for a list of our run parameters used for all our runs.



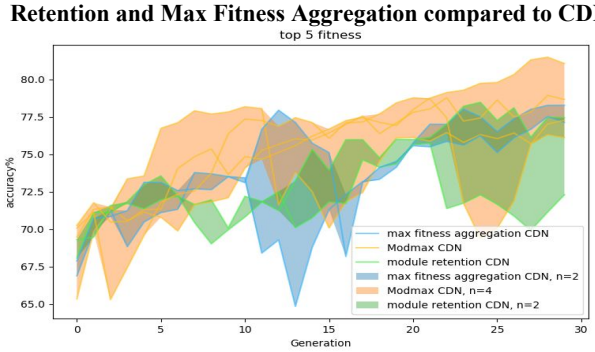
Graph 1: A graph showing the performance gains of Module Retention CDN compared to CDN



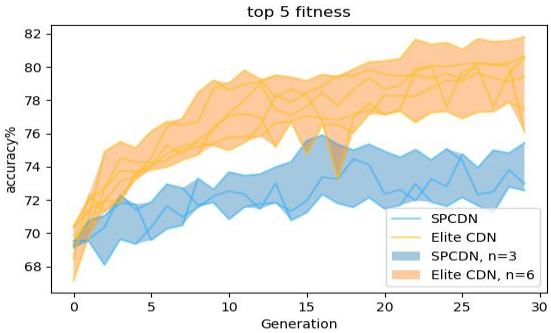
Graph 2: A graph showing the performance gains of Max Fitness Aggregation over CDN with average aggregation



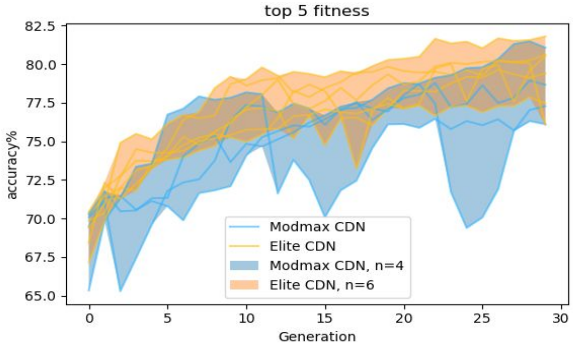
Graph 3: A graph showing the performance gains of Module Retention and Max Fitness Aggregation compared to CDN



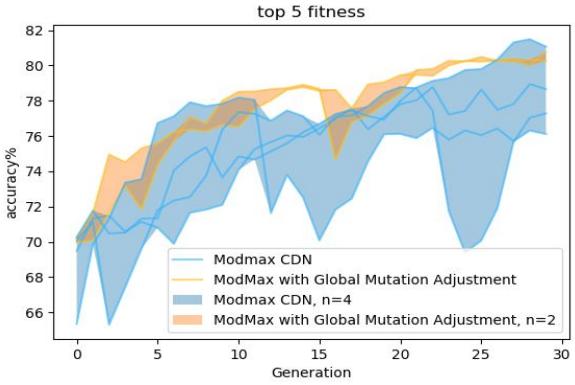
Graph 4: A graph demonstrating that the combination of Module Retention and Max Aggregation outperforms either in isolation



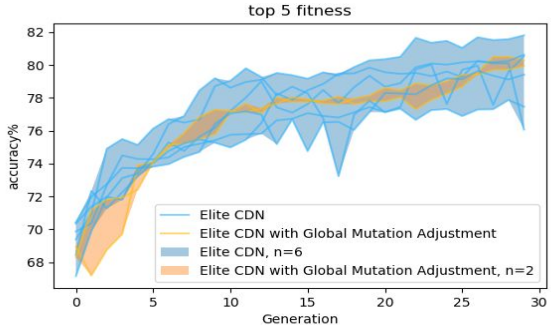
Graph 5: A graph showing the performance gains of ModMax + Speciation Overhaul (Elite CDN) over Speciation Overhaul alone



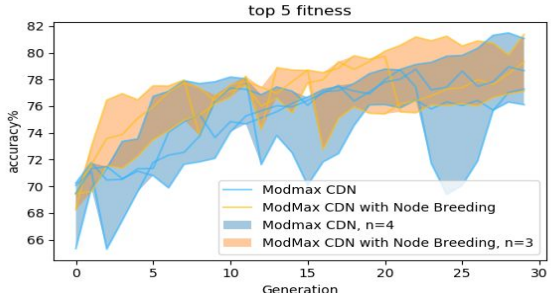
Graph 6: A graph showing the stability gains of Elite CDN over ModMax CDN



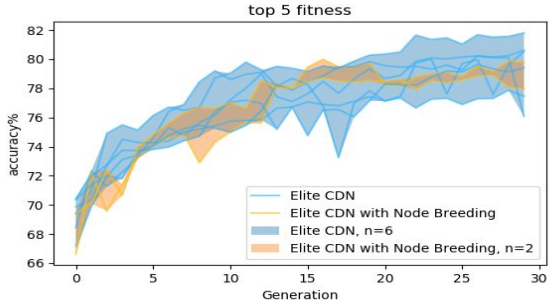
Graph 7: A graph comparing ModMax CDN against 2 runs of ModMax CDN with the Global Mutation Magnitudes Adjustment extension



Graph 8: A graph comparing Elite CDN and 2 runs of Elite CDN with the Global Mutation Magnitudes Adjustment extension



Graph 9: A graph showing the effect of NEAT Node Breeding with a 50% breed chance, when added to the ModMax CDN algorithm



Graph 10: A graph showing the effect of NEAT Node Breeding with a 75% breed chance when added to the Elite CDN algorithm

Algorithm Variation	Accuracy % in evolution	Fully trained accuracy %	2x Increased features fully trained accuracy %
Original CDN [1]	80 (12 epoch train)	92.3	-
AmoebaNet [11]	92	96.6	-
Our CDN implementation	75.83	79.88	80.6
CDN with Module Retention	79.71	-	-
CDN with Max Fitness Aggregation	79.51	-	-
ModMax CDN	81.99	84.41	85.27
Elite CDN	82.81	84.28	86.08
Elite DACDN	84.82		87.08
Elite DACDN 10 epochs	86.1	89.72	91.11
Elite DACDN Breed 20 epochs	87.69	90.66	92.12

Table 1: A table showing the maximum accuracies of differing versions of the CDN algorithm: during evolution; after fully training the evolved graph; and after fully training the evolved graph with all layer sizes doubled (feature multiplication of 2)

4.3 Discussion

4.3.1 ModMax CDN

Demonstrating that both Module Retention and Max Fitness Aggregation offer an improvement to the CDN algorithm, Graphs 1 and 2 compare their respective extensions to our implementation of the base CDN algorithm. In both cases it is visually clear that they achieve higher accuracies. Graph 1 seems to show that Module Retention allows for the CDN algorithm to raise its accuracy for longer, but not necessarily faster. Graph 2 hints that Max Fitness Aggregation allows for faster evolution. However due to the small sample sizes and the fact that this is only on one dataset, neither of these two claims can be verified. One observation which is certain however, is that both Module Retention and Max Aggregation are still not stable over time, since large drops in performance can still be observed in both extensions.

Graph 3 demonstrates that ModMax CDN offers a substantial improvement over CDN, rising both faster, and for longer. The large performance drops which can be seen indicate that ModMax, is also not stable over time. Due to this instability, we decided not to label ModMax CDN as Elite CDN. The results of Graph 4 show that ModMax outperforms both Module Retention and Max Aggregation alone. This validates our hypothesis that both Module Retention and Max Aggregation are needed together to allow successful DNNs to be recreated in later generations.

4.3.2 Elite CDN

CDN Elite is the combination of Module Retention, Max Aggregation and the Speciation Overhaul. Since the Speciation

overhaul is credited to this papers sister paper [25], we will only be demonstrating the value of ModMax, and not Speciation Overhaul itself. However Speciation Overhaul has been shown to improve the stability and performance of the CDN algorithm [25]. This is also corroborated by Graph 6. Graph 5 depicts CDN with the Speciation Overhaul being outperformed by Elite CDN. While Elite CDN can still be observed to have performance drops, it appears to be more stable than ModMax, with drops often followed immediately with recoveries. Elite CDN achieves the highest accuracy we have seen for any run of our implementation of CDN without data augmentation.

4.3.3 Global Mutation Magnitudes Adjustment

Graph 7 seems to provide positive evidence that Modmx GMA-CDN performs better than ModMax CDN. Modmax GMA-CDN outperforms ¾ of the ModMax CDN runs. Graph 8, which shows Elite GMA-CDN compared to Elite CDN, is less clear. For the majority of the generations Elite GMA-CDN's performance puts it around the average performance of the Elite CDN runs. However near the end, after the Elite CDN runs appear to have plateaued, both Elite GMA-CDN runs experience an additional rise in performance, ending up finishing near the best of the Elite CDN runs. This late stage rise in performance not typically seen in our results, may arise from the final stage of the GMA-CDN runtime where all mutations are much smaller. From both Graph 7 and 8, it appears that the GMA variations of the CDN algorithms are more consistent with each other in their performance, however due to the small sample size this cannot be stated conclusively.

GMA introduces many hand picked parameters as one must decide how quickly the mutation magnitudes drop off, by how much, and how attribute and topological mutation rates should change relative to each other. Because of a lack of time and testing resources we were unable to tune GMA beyond naive estimations. With this in mind, and due to these moderately encouraging results, we have demonstrated that a form of dynamic mutation rate adjustment would likely improve the CDN algorithm, however further testing and tuning would be required to incorporate this extension into the CDN algorithm.

4.3.4 NEAT Node Breeding

Graph 9 indicates that adding Node Breeding to ModMax CDN does not significantly affect performance. However, it appears that the accuracy rises faster, and plateaus sooner when Node Breeding is used. Graph 10 indicates that Node Breeding affects the performance of Elite CDN negatively. By generation 15, the Node Breeding runs were on par with the best of the Elite CDN runs, however at this point the Node Breeding runs plateaued, allowing Elite CDN to overtake. The runs in Graph 10 were using a breed chance of 75%, it may be that this value it too high, and due to the interpolative nature of the breeding, the attributes of the module population converged on similar values, to the detriment of exploration. It is unclear if Node Breeding pairs poorly with Elite CDN, or if it simply performs better with a lower breeding chance. Further testing would be required, however when we ran Elite DACDN [table 1] with 20 epochs per evaluation with Node Breeding both on and off, the node breeding run significantly outperformed the non Node breeding run, leading to our best result. This data while encouraging is not sufficient to make any conclusions .

4.4 Computational Time and Performance

The second best result we achieved evolved for 48 hours on 4 Tesla-V100 GPUs and achieved 91.11%. AmoebaNet which has achieved SOTA for CIFAR-10 [11] (96.6%) reported using 450 Tesla K40 GPUs for 7 days. This amounts to 390 times more GPU hours used by AmoebaNet, and while the V100 is significantly faster than the K40, this only accounts for a fraction of this deviation in evolution time. This data indicates that CDN offers a less performant but significantly faster solution to image classification NE. In Google's AmoebaNet paper [11], the authors demonstrate that AmoebaNet uses significantly fewer computational resources than other prominent automatic DNN generation algorithms used outside of NE, namely RL based architecture search [26,27]. Therefore Our claim that CDN uses significantly fewer resources than AmoebaNet extends to these RL approaches as well.

When considering this performance to computational time tradeoff, it is worth noting that our CDN implementation (without extensions) significantly underperformed when compared to the version of CDN used by the creators of CDN [1]. This is very likely due to our naive parameter choices and lack of tuning, as well as other implementation limitations not related to our extensions. Given that an excellently tuned CDN would not significantly deviate in computation time compared to a naively tuned CDN. We hypothesise that the performance gap between AmoebaNet and Elite DACDN may close somewhat, while not compromising the computational time benefits, given a well tuned Elite DACDN.

4.5 Summary

Table 1 demonstrates how much more performant the original CDN was on CIFAR-10, than our implementation, however it should be noted that the original CDN incorporated data augmentation (DA), while ours does not. Original CDN also used 8 epochs per evaluation where unless otherwise stated, we used 5.

Referring to the entries in table 1: Our CDN implementation, CDN with Module Retention, CDN with Max Fitness Aggregation, ModMax CDN, Elite CDN - it is clear that Module Retention, Max Fitness Aggregation, and the Speciation Overhaul all stack well on top of eachother. Both Module Retention and Max Aggregation increase the maximum accuracy attained in evolution by just under 4%, compared to our implementation of CDN. This puts them on par with original CDNs performance during evolution. When the two are combined they experience a further 2.5% increase in maximum accuracy, which now exceeds original CDNs performance in evolution at 81.99%. Finally When the Speciation Overhaul is stacked on ModMax CDN, becoming Elite CDN, a further 1% accuracy is gained.

Despite Elite CDN outperforming original CDN during evolution time by just under 3%, it falls far short of reaching the accuracy original CDN achieved after being trained to convergence. We hypothesise that this is partly due to a lack of DA, and partly due to our naively implemented aggregation of branched outputs

being recombined. DACDN [28] is our approach at incorporating DA into CDN, it evolves a third population (alongside blueprints and modules) of DA schemes, which are composites of simple DA operations such as mirror, or translate. The combination of this DA system and Elite CDN - Elite DACDN is our best performing CDN variant, and while achieving an impressive 86% during evolution, still fails to match original CDN when fully trained.

When comparing maximum accuracies with the original evolved DNNs against the maximum accuracies of their variants which were enlarged with feature multiplication, it is clear that feature multiplication improves the fully trained performance of these networks.

In summary, the data supports our hypothesis that ModMax CDN would exhibit behaviour more inline with an elitist EA, however it also indicates that there is more to this problem, as stability was still an issue. Introducing the stabilising properties of the Speciation Overhaul was another step towards making CDN evolutionary runs consistently improve over time. The results of Elite CDN positively confirm our research hypothesis, that the CDN algorithm can experience performance gains, we also observed that these performance gains came without significantly impacting its computational cost. The performance gains seen by comparing Elite DACDN with 10 epochs and 20 epochs (which takes twice as long) demonstrate that epoch count is an effective way to trade off computational time for performance in DACDN.

5 FUTURE WORK

5.1 CDN Tuning

A significant limitation of all of these results is the lack of time spent tuning the many CDN parameters such as population sizes, mutation rates, number of species, elite sizes, and many coefficients, including the parameters used by the extensions themselves. It is likely that given enough time, extra performance may be gained from tuning. Our results show that these extensions improved on a naively tuned CDN. Going forward we would like to spend time tuning the classic CDN parameters as well as the parameters used by our extensions.

5.2 Datasets

Throughout our testing of extensions, we limited our comparisons to CIFAR-10, a relatively small dataset in both training examples and image resolution. Furthermore the original CDN had support for recurrent networks (RNN). Since we did not implement RNNs we were unable to test our extensions in other domains such as Natural Language Processing (NLP). As future work, we would like to test our extensions on a larger image classification problem, as well as in the NLP domain.

5.3 Module Loops

This is based on an observation that some of the top DNNs in computer vision which use repeating structures, such as InceptionNet [7], also seem to repeat modules multiple times, linearly. This is to say that they instantiate multiple instances of the same module, and pass the output of each, into the input of the subsequent instance. This could be implemented in CDN with relative ease. An additional evolvable parameter could be added to each blueprint node which specifies how many times the module that node picks should be repeated. This would allow for evolving very large networks quickly, and may be very useful for larger datasets such as ImageNet.

5.4 Blueprint Node Deep Layers

Allow blueprint nodes to change type between the classic pointer to a module species from which to sample a module (as is standard in CDN [1,2]) - and a deep layer node much like in a module node. We hypothesise this would improve true elitism, as blueprints would be able to develop structures which are not dependant on the module population. We also hypothesise that allowing blueprints to ‘slot in’ small DNN structures in between its repeating structures (modules) may create a more fine grained evolutionary process where structural changes are no longer limited to changes in the repeating structures and how those repeating structures are stitched together.

6 CONCLUSION

From the data we can conclude that ModMax CDN outperforms base CDN inside the test conditions we set up. However due to the superior performance of Elite CDN over ModMax in our testing we recommend implementing all three of the proposed changes (ModMax and Speciation Overhaul) to future and existing implementations of CDN, although this certainly requires more testing. Elite CDN has succeeded in exhibiting behaviour more in line with what one would expect of an elitist EA, namely the relative absence of sharp performance drops, and consistency in improving generation by generation until convergence, without significantly increasing the computational cost. Our data also indicates that feature multiplication can help to increase the performance of evolved DNNs.

Our data indicates that the global mutation magnitude adjustment extension GMA-CDN has promise in improving the consistency and possibly performance of the CDN algorithm, however due to the limited data we cannot conclude that GMA-CDN represents a significant improvement for CDN.

While there is some evidence that NEAT Node Breeding allows for faster convergence, which could offer the potential to make CDN even faster, it is not enough to conclude that Node Breeding should be included in future implementation of CDN. Further experimenting and tuning would be required to properly assess the value of Node Breeding.

Despite our implementation of CDN being inferior to the original implementation [1,2], we hypothesise that the performance gains seen from our extensions would transfer to a better designed and tuned CDN implementation. Our extensions, namely Elite CDN and its DA counterpart Elite DACDN, may represent progress in increasing the performance of the relatively low performing, but notably fast CDN NE algorithm, thus closing the performance gap somewhat between the moderately computationally expensive (CDN) and the highly computationally expensive (RL, AmoebaNet) architecture search algorithms. We believe this is a small step in democratising NE and AI in general.

7 REFERENCES

[1] Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N. and Hodjat, B., 2019. Evolving deep neural networks. In Artificial Intelligence in the Age of Neural Networks and Brain Computing (pp. 293-312). Academic Press.

[2] He, Y., Lin, J., Liu, Z., Wang, H., Li, L.J. and Han, S., 2018. Amc: Automl for model compression and acceleration on mobile devices. In Proceedings of the European Conference on Computer Vision (ECCV) (pp. 784-800).

[3] Ronan Collobert and Jason Weston. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In Proceedings of the 25th international conference on Machine learning. ACM, 160–167.

[4] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In 2013 IEEE International Conference on Acoustics, Speech and Signal Processing. IEEE, 6645–6649.

[5] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J. and Wojna, Z., 2016. Rethinking the inception architecture for computer vision. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2818-2826).

[6] El-Sawy, A., Hazem, E.B. and Loey, M., 2016, October. CNN for handwritten arabic digits recognition based on LeNet-5. In International Conference on Advanced Intelligent Systems and Informatics (pp. 566-575). Springer, Cham.

[7] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1-9).

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

[9] He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

[10] Xie, S., Girshick, R., Dollár, P., Tu, Z. and He, K., 2017. Aggregated residual transformations for deep neural networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1492-1500).

[11] Real, E., Aggarwal, A., Huang, Y. and Le, Q.V., 2019, July. Regularized evolution for image classifier architecture search. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 33, pp. 4780-4789).

[12] Meyer-Lee, G., Uppili, H. and Zhuolun Zhao A. 2018. Evolving Deep Neural Networks. [online] Retrieved August 28 2019 from

[13] Hornik, K., Stinchcombe, M. and White, H., 1989. Multilayer feedforward networks are universal approximators. Neural networks, 2(5), pp.359-366.

[14] W. Xing. (2018) Deep learning diagram [online] Researchgate Available at: https://www.researchgate.net/Figure/Deep-learning-diagram_fig5_323784695 [Accessed 31 August 2019]

[15] LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P., 1998. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), pp.2278-2324.

[16] Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105), 1-3

[17] S. Whiteson. (2012) The basic steps of neuroevolution [online] Researchgate Available at: https://www.researchgate.net/Figure/The-basic-steps-of-neuroevolution_fig1_228529570 [Accessed 31 August 2019]

[18] Stanley, K.O. and Miikkulainen, R., 2002. Evolving neural networks through augmenting topologies. Evolutionary computation, 10(2), pp.99-127, 1-12 22-24

[19] Abbass, H.A., Sarker, R. and Newton, C., 2001, May. PDE: a Pareto-frontier differential evolution approach for multi-objective optimization problems. In Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No. 01TH8546) (Vol. 2, pp. 971-978). IEEE.

[20] Deb, K., Pratap, A., Agarwal, S. and Meyarivan, T.A.M.T., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE transactions on evolutionary computation, 6(2), pp.182-197.

[21] Deb, K. and Goel, T., 2001, March. Controlled elitist non-dominated sorting genetic algorithms for better convergence. In International conference on evolutionary multi-criterion optimization (pp. 67-81). Springer, Berlin, Heidelberg.

[22] Zitzler, E., Deb, K. and Thiele, L., 2000. Comparison of multiobjective evolutionary algorithms: Empirical results. Evolutionary computation, 8(2), pp.173-195.

[23] Thierens, D., 2002, May. Adaptive mutation rate control schemes in genetic algorithms. In Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600) (Vol. 1, pp. 980-985). IEEE.

[24] A Auger. 2009. Benchmarking the (1+1) Evolution Strategy with One-Fifth Success Rule on the BBOB-2009 Function Testbed. ACM-GECCO Genetic and Evolutionary Computation Conference, Jul 2009, Montreal, Canada. Inria-00430515f

[25] Abramowitz, S.. 2019. Improving CoDeepNEATs performance through intelligent speciation . Honours. University of Cape Town. Cape Town

[26] C. Liu, B. Zoph, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. 2018. Progressive neural architecture search. ECCV, 2018.

[27] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. 2018. Learning transferable architectures for scalable image recognition. In CVPR, 2018.

[28] Toledo, L. 2019. Improving CoDeepNEAT's performance using co-evolved data augmentations. Honours. University of Cape Town. Cape Town

8 APPENDIX

8.1 Parameters Tables

Static Evolution parameter	Value
Module population size	56
New Module node mutation chance	8%
New Module connection mutation chance	8%
Module Connection enable/disable mutation chance	5%
Target number of module species	4
Blueprint population size	22
New Blueprint node mutation chance	16%
New Blueprint connection mutation chance	12%
Blueprint Connection enable/disable mutation chance	5%
Percent to reproduce	20%
Elite to keep	10%
Evaluations per generation	75
Deep Layer size mutation chance	22%
Deep Layer type mutation chance	8%
Number of generations	[30,50]

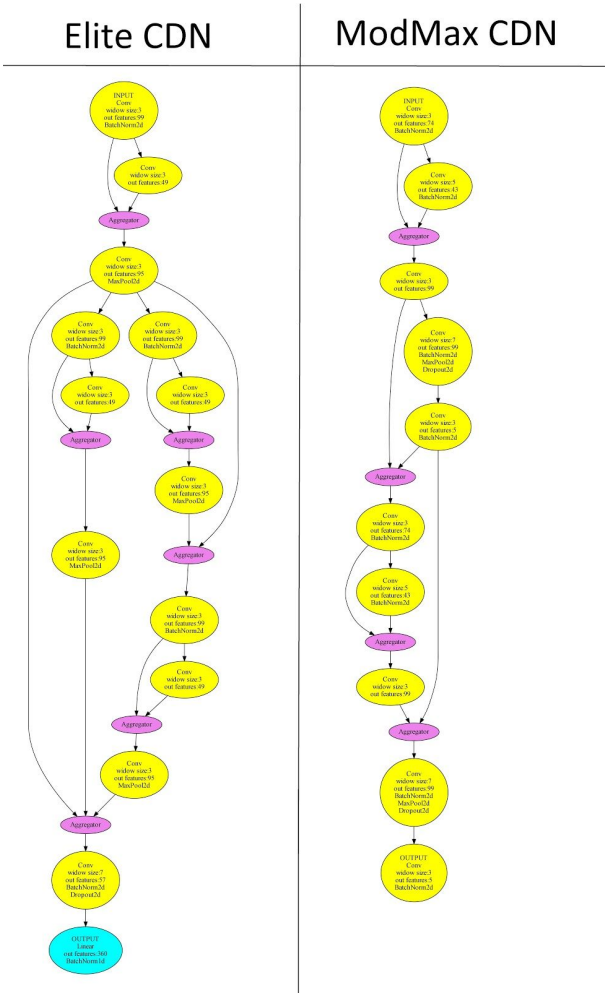
Table 2: The static parameters used during our CDN evolutionary runs

Dynamic Evolution function name	Function	Description
Module mapping ignore chance	$\frac{Number\_of\_maps^{1.5}}{Number\ of\ species^{1.5}}$	Chance to ignore a single mapping when parsing blueprint to DNN. Used by Module Mapping ignores [Section 3.1.3]
Number of maps	-	The number of unique species_numbers the blueprint is mapping to a module
Number of species	Number_of_species >= number of maps	The total number of unique species_numbers in the blueprint
Run frac	$\frac{Generation\_number}{numer\ of\ generations}$	How far the run is as a fraction
Global attribute mutation modifier	$\frac{\arctan(6.2 - 10*run\_frac)+3.6}{4}$	Function to smoothly drop off the mutation rate of module node attributes - more slow drop off than topology modifier. Used by GMA-CDN [Section 3.3.2]
Global topology mutation modifier	$\frac{\arctan(3 - 7.5*run\_frac)+3.6}{3.5}$	Function to smoothly drop off the mutation rate of module node attributes - faster drop off than attribute modifier. Used by GMA-CDN [Section 3.3.2]

Table 3: The dynamic parameters used during our CDN evolutionary runs. These are values which are functions of other functions.

8.2 Graphs

This section contains the top performing DNNs created by various CDN runs. It should be noted that after each one of the graphs depicted here, a final linear layer is added which maps the dimensions of the output nodes output features to the dimension of the target function. Yellow nodes denote convolutional layers while blue nodes denote linear/fully connected layers.



Graph 11: The best DNNs Developed by Elite CDN and ModMax CDN with Convolutional layers (bellow) and Linear/fully connected layers (blue)

8.3 Additional Material

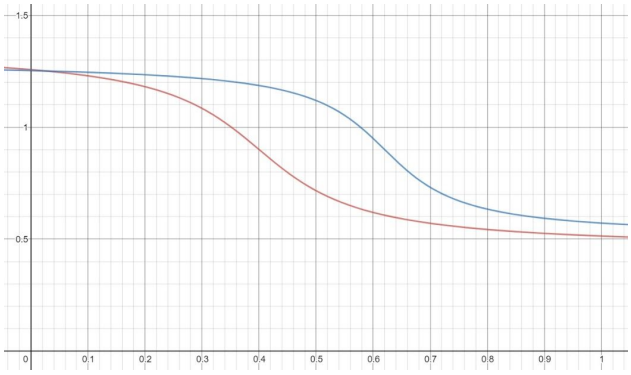


Figure 5: Attribute (Blue) and Topology (Red) global mutation modifiers over % progress in evolutionary run. This diagram depicts the smooth drop off of the topological and attribute mutation rates, as well as the divergence between the two