Improving CoDeepNEATs performance through intelligent speciation

Sasha Abramowitz
Computer Science
University of Cape Town
Cape Town, South Africa
reallysasha@gmail.com

ABSTRACT

CoDeepNEAT is a Neuro-Evolution algorithm that performs network architecture search and hyper-parameter tuning for deep neural networks. It attempts to mimic modern state-of-the-art deep neural networks by evolving repeating structures. The goal of this paper is to improve the performance of CoDeepNEAT through an improved speciation procedure. We propose SPCDN which performs intelligent species representative selection and adjusts how an individual joins a species. These modifications showed an improvement in the stability of the CoDeepNEAT algorithm, when tested on the CIFAR-10 dataset. The results show that our implementation of CoDeepNEAT evolves performant deep neural network structures, faster than other notable Nuero-Evolution algorithms.

1. INTRODUCTION

Motivated by the improvements in computing power and quality and volume of available data, machine learning has obtained best in class performance in computer vision, natural language processing and speech recognition [18, 19, 20]. Specifically deep neural networks have achieved state-of-the-art results when given enough compute time and data resources to learn useful and generalisable patterns in complex datasets. However modern state-of-the-art DNNs are incredibly large and rely on domain experts to design their architectures. This led to a rise in popularity of automatic machine learning (autoML) because it is well suited to performing this extensive tuning. AutoML is able to design the structure and hyper-parameters of a DNN with little human intervention [6].

CoDeepNEAT [1, 2], which is the focus of this paper, is an autoML algorithm. CoDeepNEAT attempts to mimic the trend of highly performant, hand-crafted DNNs by using repetitive sub structures [5]. However CoDeepNEAT has its limitations, specifically in the way it handles elitism and speciation. True elitism does not exist in CoDeepNEAT, because of the way artificial neural networks (ANNs) are assembled. There is randomness involved in the assembly process, thus an individual in generation *n* may not be assembled the same way in generation *n+1*. The lack of true elitism [15] leads to the best ANNs being *lost* in subsequent generations, which is highly undesirable. CoDeepNEAT borrows its speciation strategy from NEAT [3] and while the speciation strategy works well for NEAT, species in CoDeepNEAT need to constantly represent the same or similar functional niche. If the functional niche of a species continuously changes it will hinder CoDeepNEATs ability to create and evolve improved ANNs, because individuals rely on the functional niche species represent.

Therefore we propose modifications to speciation, which we hypothesize will allow the functional niche of a species to fluctuate less across generations. These modifications will change the way CoDeepNEAT chooses species representatives, how individuals join a species, the distance formula and how ANNs are assembled.

1.1 Research question

We hypothesise the modifications proposed in section 3 will have a significant impact on the stability of the evolutionary process of CoDeepNEAT, which will lead to an increase in the maximum accuracy achieved.

2. BACKGROUND

This section explains the CoDeepNEAT algorithm in detail, automated machine learning (autoML) and deep learning in general.

2.1 Deep Learning

Deep learning refers to the creation of ANNs with multiple hidden layers. A common type of deep learning architecture is a convolutional neural network (CNN), which is commonly used for computer vision tasks. In a CNN, images are initially passed through convolutions, so that they are downsized, then passed through fully connected layers, which is visualized in figure 1. CNNs are currently state-of-the-art in the domain of image recognition [16].

2.2 Automated machine learning

CoDeepNEAT [1, 2] is an autoML [10] algorithm, more specifically CoDeepNEAT performs a network architecture search (NAS) [11] and hyper-parameter optimisation [9] using an evolutionary approach, making it a Neuro-Evolution (NE) [12] algorithm. NE is a subset of autoML that specifically use evolutionary algorithms to build ANNs. NAS automatically tunes an ANN architecture so that it improves in some metric, usually

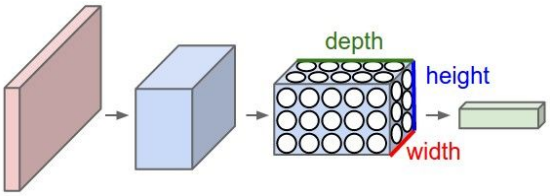

Figure 1: The CNN shows how the image is downsized and reshaped from the convolution operations.

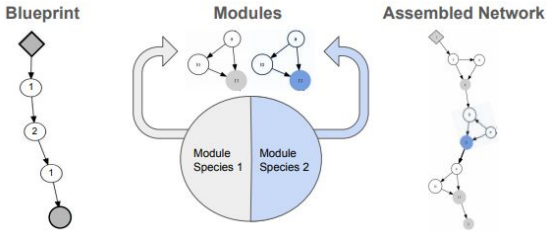

Figure 2: Depiction of the process of converting blueprints into assembled ANNs, by sampling from module species [1].

accuracy. NAS generally changes the topology of an ANN by adding or removing entire layers and increasing or decreasing the size of a layer. Hyper-parameter optimization tunes the *options* (including learning rate, dropout rate and other hyper-parameters) of an ANN. The most naive hyper-parameter search is grid search [9], this algorithm tests ANNs with every combination of every option.

2.3 CoDeepNeat

CoDeepNEAT is based on NEAT [3]. In NEAT the weights and topologies of a population of ANNs are evolved using an evolutionary algorithm. On the other hand CoDeepNEAT uses NEAT to evolve the topologies of ANNs and makes use of backpropagation to train ANNs. The test accuracy is used as the fitness value, which is used to decide which ANNs reproduce and survive into the next generations.

The easiest way to understand CoDeepNEAT is first to understand DeepNEAT [2]. DeepNEAT follows the NEAT algorithm almost exactly: a population of individuals (ANNs) with minimal complexity are created. The topologies of the individuals are changed via mutation. Historical markings are used to assist in crossover. The individuals are divided into species according to a similarity metric. A better performing species grows larger than a poorly performing species. To protect innovation an individual only directly competes with other members of its species. All of the above happens in both NEAT and DeepNEAT, however, in DeepNEAT the representation of the ANN (genotype) greatly differs from the actual ANN (phenotype). A node in the genotype represents a DNN layer in the phenotype. This allows the standard NEAT algorithm to evolve the genotype while weight training and evaluation can be carried out using backpropagation on the phenotype. The fitness passed back to the genotype is the accuracy obtained during the training of the phenotype.

1. **Given** population of modules/blueprints
2. **For each** blueprint B_i during every generation:
    3. **For each** node N_j in B_i
        4. **Choose** randomly from module species that N_j points to
        5. **Replace** N_j with randomly chosen module M_j
    6. **When** all nodes in B_i are replaced, convert B_i to assembled network N_i
7. **Evaluate** fitnesses of the assembled networks N
8. **For each** network N_i
    9. **Attribute** fitness of N_i to its component blueprint B_i and modules M_j
10. **Evolve** blueprint and module population with NEAT

Algorithm 1: Evolution and training procedure followed by CoDeepNEAT [1].

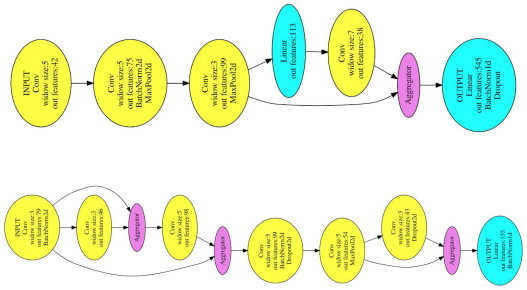

Figure 3: Top: the best blueprint with inserted modules, created by SPCDN. Bottom: the best blueprint with inserted modules, created by SPCDN with GED.

Coevolution is the difference between DeepNEAT and CoDeepNEAT. This coevolution happens between two populations of NEAT individuals, one of blueprints and one of modules. Blueprints represent how repeating structures are connected, while modules represent the repetitive sub structures and hold parameters needed to build the ANN. A phenotype (ANN) is made from the blueprint and the modules that the blueprint chooses to create the ANN, two examples of this can be seen in figure 3. A blueprint chooses these modules by randomly sampling modules from module species. This random sampling has two effects: each time a phenotype is created it is created differently and modules have the chance to not be sampled and thus not evaluated in a generation. To combat this second issue blueprints are evaluated multiple times, sampling new random modules each time, increasing the probability that modules will be evaluated. Modules and blueprints will receive a fitness as the average accuracy of the ANNs they were used to create.

The exact process to assemble a phenotype (steps 3, 4 and 5 in algorithm 1 and visualized in figure 2) is as follows: each node in a blueprint holds a reference to a species in the module population. When assembling a phenotype every node in the blueprint is replaced by a randomly selected module in the species that the node points to. Every node in the module holds all the necessary information to convert itself to an ANN layer (that is, layer type, layer size, kernel size, activation function and so on) and its connections dictate how inputs and outputs are passed between nodes. Once all blueprint nodes are replaced by modules, the modules are converted to small ANNs which are connected to each other through the blueprints connections. Every node in the blueprint that points to the same species is forced to pick the same module, however each time the blueprint is sampled this modules are sampled again. This rule allows

for repeating structures to occur, which is a key to the success of CoDeepNEAT. As demonstrated by Szegedy, C et al. [5] these repetitive structures occur often in many state-of-the-art hand crafted DNNs.

CoDeepNEAT [1] tries to optimize ANNs not only for accuracy, but also for size. It achieves this by using a multiobjective [14] approach to find the pareto front [13] when ranking ANNs. The pareto front is the group of all solutions that are not worse than any other solution in all metrics (accuracy and size in this case). It does this not only to keep ANNs small and performant, but also because this was found to improve overall accuracy [1]. It also forces each generation to try and achieve better results with smaller ANNs instead of blindly increasing the layer size or adding more layers.

3. Method

This section outlines the limitations of CoDeepNEAT and the modifications made to combat them.

3.1 CoDeepNEAT limitations

To understand why we made the modifications that we did, one must first understand the issues that are present in CoDeepNEAT.

3.1.1 Species stability

The first flaw of CoDeepNEAT lies in how reliant blueprints are on module species being stable (continuously representing the same functional niche). This is because the only attribute that a blueprint node holds is a reference to a module species and thus is entirely reliant on those species being stable. For a species to be stable it must represent the same or similar types of ANNs, meaning that they have a similar functional niche, from generation to generation. For example if in generation one a species represents small convolutional ANNs, it should not represent large fully connected ANNs in generation two. If this rapid functional change did occur blueprints would not be able to co-evolve with modules, since the functions of the species change faster than the blueprint nodes can change in response.

Speciation in CoDeepNEAT follows NEAT speciation, which uses a representative system to decide which individuals join a species. This system works well for NEAT, since the job of a species in NEAT is to protect innovation [3], however in CoDeepNEAT a species is also tied to the functional niche that its members represent. In NEAT and CoDeepNEAT an individual is added to a species if it is similar enough to that species' representative, where similarity is defined by a function on the number of dissimilar connections of each genome (formula 1). Representative selection happens at the end of every generation, where a new individual is chosen at random from each species to be that species' representative. Therefore if this random representative does not adequately represent all the individuals in its species, meaning it does not reflect the general function of the species' other members, it may alter which individuals join the species in the next generation, allowing functionally dissimilar members in, thus altering the functional niche which the species represents. This species would then not be stable since its functional niche has changed in one generation.

The lack of a representative that promotes species stability for a particular species leads to issues with blueprint nodes since they are defined by the species they point to. Therefore if individuals in a species can change drastically from generation to generation, because of random representative selection, the species that a blueprint node points to greatly loses its significance, since the functional niche of the species that it was pointing to has changed. The loss in significance inhibits the blueprints ability to meaningfully optimize the ANN in the long term, because the sub-structures to which its nodes point are constantly changing, thus not allowing it to properly search for better topologies.

3.1.2 Elitism

The second limitation of CoDeepNEAT lies in its elitism, or lack thereof. Elitism is the process of keeping the fittest solutions alive into the next generation and has been shown to greatly improve evolutionary algorithms in many tasks domains [15]. True elitism allows a generation's most fit solution/phenotype to be at least as fit as the best solution in the previous generation. A form of elitism is included in the CoDeepNEAT algorithm, however because of the way modules are selected it does not perform as desired. An elite blueprint in generation *n* is unlikely to pick the same modules as it picked generation *n-1*. This is caused by three factors. First, elitism expects the best solutions to continue on to the next generation unchanged, however the elite blueprints do not necessarily pick the same modules in the next generation, since each module is selected randomly from a species. Second, the modules in an elite blueprint may not be elite modules and thus would not survive to the next generation, therefore not giving an elite blueprint the option to select them in the next generation. Third, since species can be unstable an elite blueprint that performed well in the last generation may perform poorly in the next, since the species that it was pointing to no longer represents the same functional niche.

3.2 Speciation modifications

This section includes all modifications made to the speciation process of CoDeepNEAT. The modifications made to speciation are an attempt to keep the species more stable, thus allowing each blueprint node to lock on to a species which represents a fairly consistent functional niche from generation to generation.

3.2.1 Reducing speciation randomness

The first modification changes the way species are assigned. In NEAT an individual (ANN) is placed in the first species whose representative (which defines the species via similarity) is adequately similar to the individual. This presents two problems, first, it biases species towards the beginning of the species list since an individual joins the first similar enough species. Second, it can lead to two species representing very similar

niches, since individuals don't necessarily join the species that they are most similar to, there might be a better suited species for an individual. The modification places an individual in the species with a representative that it is most similar to. This should aid species in having a concrete representation while also not biasing any species to be more popular.

Our second modification aims to improve species representative selection. The representative selection in NEAT is quite naive and not well suited to CoDeepNEAT. NEAT picks a new random representative, from the pool of current species members, for each species, at the end of a generation. If the new representative is an individual that is not adequately similar to the rest of the species, then this species' functional niche will change in the next generation. Therefore we propose picking a representative that is most similar to each member of the species. This is achieved by obtaining the similarity (formula 1) of each member to each other member of the species (as a number) and choosing the member with the smallest summed distance. This was likely not implemented in NEAT because it is quite computationally expensive, however when compared to training an ANN it does not take nearly as much time and thus is negligible. Conceptually this will allow species to represent the same or a very similar niche from generation to generation.

These two modifications on top of CoDeepNEAT will henceforth be referred to as SPCDN.

3.2.2 Graph similarity

In NEAT an ANN is represented as a graph and while the current topological comparison used in NEAT works very well, a more mathematically based graph comparison algorithm might be able to improve upon its performance. NEAT compares graphs by applying a formula on the number of connections that the graphs do not share, specifically by the equation:

$$d = \frac{C_1 \times D + C_2 \times E}{N} + W$$

Formula 1: Graph similarity as defined by NEAT

Where C_1 and C_2 are coefficients, D is the number of disjoint connections [3], E is the number of excess connections [3], N is the number of connections of the longest individual and W is the average weight difference [3] of common connections. Conceptually this equation works very well, since the more dissimilar connections a graph has, the higher the final value and thus the more dissimilar the graphs are. However it does require some tuning specifically regarding the coefficients. Therefore a possibly better solution is to find the graph edit distance (GED) [20]. This has been shown to be a good metric for comparing graphs and it does not require any tunable parameters. Simply the GED between two graphs g_1 and g_2 is the number of edits that must be applied to g_1 for it to become g_2. This distance is likely a better measure of similarity than NEAT distance (formula 1) and therefore it should more precisely speciate individuals based on topology than NEAT distance.

3.2.3 Attribute similarity

Formula 1 shows how similarity is defined within NEAT, however CoDeepNEAT does not have any use for the average weight difference since connections do not hold weights in CoDeepNEAT. However, a node in a CoDeepNEAT holds attributes such as layer size and layer type. This modification attempts to make a meaningful change to the similarity formula in order for the genomes to take into consideration layer type and size. This allows individuals to be differentiated by more than just topology which should allow the similarity metric to make more fine grained decisions and link the attributes of a node to the species functional niche. This modification was inspired by an online implementation of CoDeepNEAT [4], which achieved good results when using the attribute similarity metric.

The new metric acts the same as the weight term in the original algorithm, by adding a normalized difference in layer size if layers are the same type (that is, both are convolutional layers) or adding a constant value if layer types are different. This new term is then divided by the number of nodes of the largest genome to get the average similarity per node. This modification does present a large limitation in that it would require a lot of tuning to properly balance the topology and attribute similarity in a meaningful way.

3.2.4 Changing blueprint node identity

This modification changes CoDeepNEAT at a much more fundamental level than any of the previous modifications. It modifies the blueprint nodes identity by changing blueprints nodes from holding a reference to a species to holding a representative (which is a module). This allows a blueprint node to act as its own species. Blueprint nodes use their representative in the same way a species does, by gathering the most similar individuals to that representative. This system allows very fine grained control and bypasses the need for a blueprint node to hold a species reference.

Using this modification blueprint nodes pick a module node randomly (weighted by similarity) from the n most similar nodes to its representative. The representative can be mutated in four different ways, it can be reselected randomly from: the entire module population, the n most similar nodes, the n least similar nodes and all other representatives used in the blueprint. These options all have a purpose, choosing from the entire population and choosing from the n furthest individuals allows the space of possible representatives to be better explored. Choosing from the most similar representatives allows for slight adjustments to be made. Choosing from the other representatives in the same blueprint allows for the repeating structures which are crucial to CoDeepNEATs success. Another way repeating structures are preserved is by allowing the mutation of a single representative to affect all other representatives in the same blueprint. This acts as a way to replace a single repeating structure with a different one at every single position in the ANN.

This modification does not come without a cost. It will likely not be as good as original CoDeepNEAT at

repeating structures, however it does give blueprints a much more fine grained representation.

This modification will henceforth be referred to as REPCDN.

3.3 Module retention modifications

3.3.1 Fixing elitism

This modification aims to promote elitism in CoDeepNEAT. The main issue with elitism in CoDeepNEAT is that blueprints will be randomly assembled using different modules each generation. Module retention addresses this by allowing the elite blueprints to remember their modules if possible. In the ideal case this allows the top performing blueprints to hold onto all of their modules and not have to randomly select a module each generation. This leads to fewer drops in the top accuracy between generations.

3.3.2 Maximum aggregation

We decided to take this module retention idea further by changing the strategy for aggregating multiple evaluations of a single individual, since both blueprints and modules are sampled multiple times in a single generation. In original CoDeepNEAT [1, 2] if a module or blueprint is evaluated multiple times it receives the mean fitness of all evaluations. We decided to change this to the maximum fitness of all evaluations. This allows the best performing blueprint with all of its modules to live until the next generation. Blueprints can use modules in multiple ways, they can put them near the beginning or end of the ANN and they can use them frequently or only once. Knowing this we implemented maximum aggregation to penalize blueprints that use modules in an incorrect way, without penalising the modules it used badly. For example if blueprint *X* obtained a low accuracy, but blueprint *Y* obtained a high accuracy and both *X* and *Y* used the same modules, those modules should not be penalized for a poor performance in blueprint *X*. Simply modules should not be penalized for being used incorrectly and blueprints should not be penalized for choosing modules incorrectly.

The combination of the modifications in 3.3.1 and 3.3.2 will henceforth be referred to as modmax.

3.3 Deviations

This section outlines any minor changes made to the CoDeepNEAT algorithm or implementation details that were not specified in the original papers [1, 2].

Feature multiplication is a minor deviation made to increase the speed of CoDeepNEAT. It works by multiplying all layer size by more than 1 when fully training the ANN. This allows for proportionally larger ANNs during fully training to aid in achieving a higher accuracy.

Two important details not mentioned in either original CoDeepNEAT papers [1, 2] are starting population and speciation strategy. The starting population consists of two topologies, a triangular and a linear topology. This was done because NEAT works best if the starting population is a simple and these topologies are the most simple possible, containing at most three nodes. The speciation strategy used was dynamic speciation. This strategy aims for *n* species by linearly increasing and decreasing the speciation threshold. This threshold is the distance an individual must be from the species representative in order to join that species (see table 2 for the target number of species).

A very important and unmentioned detail in the original papers [1, 2], is whether or not CoDeepNEAT used a supplementary similarity metric, such as the suggested modification in section 3.2.3. A key feature of NEAT is its ability to calculate the similarity of two ANNs. However, in NEAT, unlike in CoDeepNEAT, nodes do not represent layers. Therefore some kind of supplementary similarity metric that takes into account layer type and size would likely be a useful addition. This was not included in our base CoDeepNEAT since it was not specified in either paper. However, an online implementation of CoDeepNEAT [4] did something similar. Our implementation does not use the same method as Meyer-Lee G. et al. [4], but the idea is the same.

Original CoDeepNEAT does perform some data augmentations. The hyper-parameters of these augmentations belong to a blueprint and thus are optimized by the evolutionary algorithm. However we did not implement this because data augmentations would make the training process much slower, since many images needs to be augmented, which is an expensive process.

A significant deviation from CoDeepNEAT is species shuffling. This was implemented because we observed that since the initial population consists of only two topologies, the total number of species created in the first generation was usually two whereas our target is four (table 2). This is an issue because blueprint nodes can only choose from the available species. Once they have made the initial random choice from the available species, it takes future generations a long time to properly explore new species which arise. We solved this by giving each blueprint node a high chance (table 2) to change their species number each generation until the target number of species is reached. This allowed species to have a good distribution and therefore all modules had a fair chance at being evaluated.

3.5 Experimentation

Given that CoDeepNEAT is not open source these modifications were built upon our own implementation of CoDeepNEAT. It was built with pytorch[1], using the details from the original CoDeepNEAT papers [1, 2] and is available on GitHub[2]. Therefore even if our base CoDeepNEAT does not achieve the same results as in [1, 2] if a modification improves upon our implementation,

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

it is likely that it will improve any implementation, at least to some degree.

We decided to benchmark all of our modifications on the CIFAR-10[3] dataset. This was also done in the original implementation [2] and thus allowed an easy comparison. The original CoDeepNEAT that used CIFAR-10 as a benchmark, was single-objective, thus we decided to conduct all of our tests using a single objective version of CoDeepNEAT to allow fair comparison to the original. Modifications were tested in isolation (against our implementation of the original) and combined with each other. This was done to help reduce the randomness and to see if modifications could be used together to obtain even better results.

Multiple runs with the same parameters were done in order to explore the variance that a particular modification produced. Thus most graphs include an area and multiple line graphs (see figure 4). This area represents the distance between the maximum and minimum values of all runs of an experiment with the same parameters.

4. Results and discussion

This section analyses and explains the graphs we obtained from our experiments. All graphs, except figure 7, depict experiments with the same parameters (table 3), however the 2 experiments being compared will always differ by a single modification. In the key of each graph (except figure 7) there is a parameter n, which refers to the number of times each experiment was repeated in order to guarantee the accuracy of the results. All of the graphs (except figure 7) plot the average of the top 5 accuracies achieved during evolution at each generation. This was done to avoid some of the randomness introduced by only training for 5 epochs.

Table 1 depicts the accuracy of an ANN before and after fully training. When fully training, the size/number of features of each layer of the ANN was doubled to allow it to make better use of more epochs.

Figure 7 was obtained by sampling 20 random individuals from each generation and splitting them into two groups of 10. Both the NEAT and graph edit distance were found from each individual in one group to each individual in the other group. These values were obtained for 41 generations and then normalized by their mean. The variance of the normalized distance values was found at each generation for each different distance metric and this was plotted. The graph shows GEDs ability to differentiate graphs better as they get more complex compared to NEAT dist. A higher value means that more individuals were found to be topologically different and a lower value means that many individuals were found to be topologically similar.

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

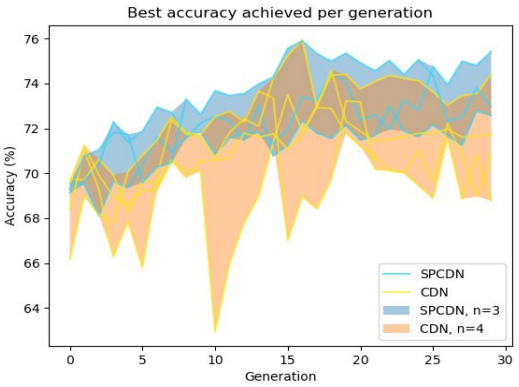

Figure 4: This graph compares SPCDN to our implementation of CoDeepNEAT.

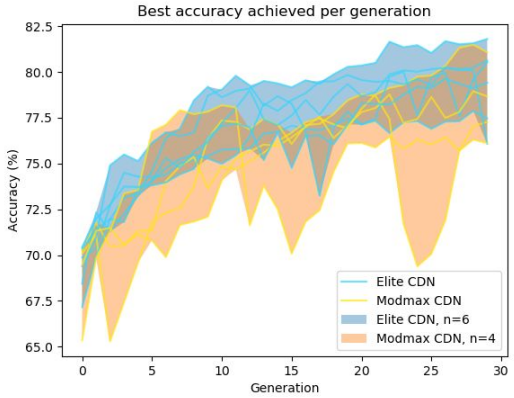

Figure 5: A graph showing the stability and accuracy of elite CoDeepNEAT over modmax. Elite CoDeepNEAT is the combination of modmax and SPCDN.

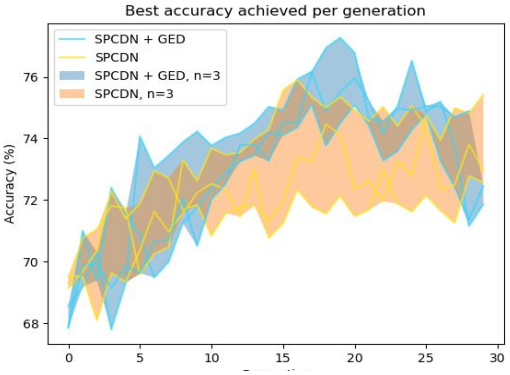

Figure 6: A graph comparing SPCDN to SPCDN using graph edit distance.

Experiment	Accuracy in evolution	Accuracy after fully trained
Original CoDeepNEAT [1, 2]	80% (12 epochs) [2]	92.3% [2]
Our CoDeepNEAT implementation	75.83%	80.6%
SPCDN	76.71%	82.25
SPCDN + GED	78.04%	82.12%
Elite CDN	82.81%	86.15%
Elite DACDN (10 epochs)	86.1	91.11
DA Elite CDN + node breeding [7, 8] (20 epochs)	87.69%	92.12%

Table 1: This shows our results obtained for multiple experiments. Column 1 is the best accuracy achieved during evolution (low epochs). Column 2 is the accuracy after fully training an ANN for 300 epochs.

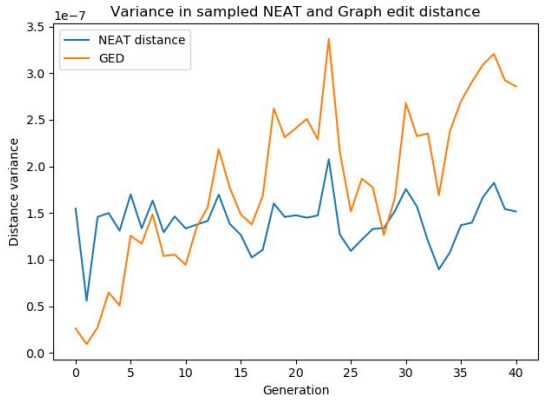

Figure 7: A graph showing the variance in distance for both NEAT and graph edit distance. NEAT graphs get more complex in later generations, thus this represents the distance functions ability to differentiate graphs as the complexity increases. A higher value means that more individuals were found to be topologically different.

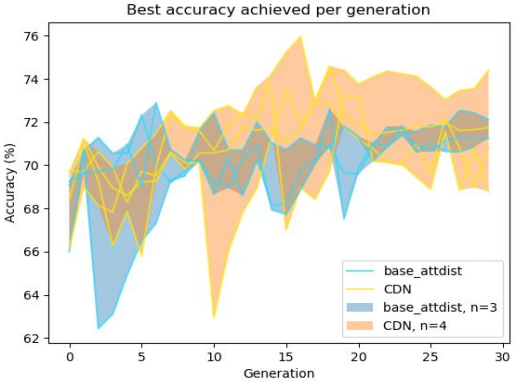

Figure 8: A graph comparing original CoDeepNEAT to CoDeepNEAT using attribute distance.

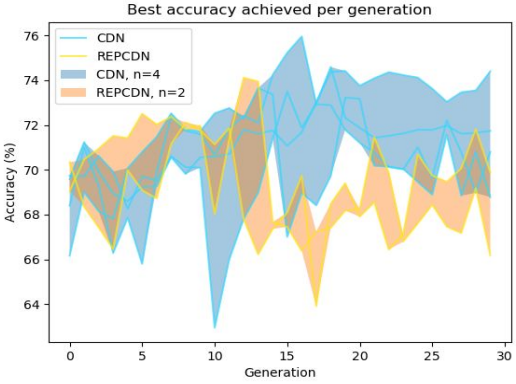

Figure 9: A graph showing the performance of REPCDN when compared to original CoDeepNEAT.

4.1 Discussion

4.1.1 Better representative selection

SPCDN (section 3.2.1) has not only improved on our implementation of original CoDeepNEAT [1, 2] (figure 4), but also on our modmax (section 3.3) modification (figure 5). This improvement is not only seen in the accuracy gains, but also the stability and run to run consistency of SPCDN.

SPCDN outperforms our implementation of original CoDeepNEAT, while elite CDN (which combines SPCDN with modmax (section 3.3)) outperforms modmax. This shows that the inclusion of the speciation modification improves the accuracy of CoDeepNEAT and works well with other modifications. With SPCDN achieving a maximum accuracy of 76.71% when compared to base CoDeepNEATs max accuracy of 75.83%. While elite CDN achieved a max accuracy of 82.81%, compared to modmax's best accuracy of 81.99%. This also shows that the inclusion of SPCDN produces slightly more accurate ANNs. The higher accuracy modmax achieved is discussed further by Acton, S [7].

The biggest benefit of SPCDN is that it allows for a more consistent rise in accuracy over the generations. This consistency stems from SPCDNs ability to remove the most likely cause of the large drops in accuracy, which can be seen in figures 4 and 5 from the base and modmax graphs - for example figure 4 generation 10. The likely cause of these drops is a large change in the functional niche of one or more species. This change would force blueprint nodes to pick undesirable modules thus leading to a drop in accuracy. Since SPCDN is built to fix the changing in functional niches of a species it avoids the large drops in accuracy, therefore allowing for a more consistent rise in accuracy.

A minor benefit of SPCDN is that runs are more consistent with each other. This is demonstrated by the much thicker band that the base graph creates when compared to SPCDN (figure 4).

4.1.2 Graph edit distance

Taking figure 6 at face value, one could say that GED does improve upon speciation, however upon closer inspection it does not offer a huge improvement. For most generations GED achieves a slightly higher accuracy than SPCDN, however it is never much more than 0.5% more accurate. This could be attributed to randomness caused by the low number of training epochs. When looking at the highest achieved accuracy, within 5 epochs, GED achieved 78.04% while SPCDNs maximum accuracy was 76.71%. This again shows that GED is slightly better than the simple NEAT distance formula. The explanation for this can be found in figure 7. It shows that the variance of the GED distances increases as graphs get more complex, while NEAT distance stays relatively constant, meaning that GED is able to better distiguish complex graphs than NEAT distance.

Couple the performance improvements with the fact that GED removes the need for two hyper-parameters, which are required for NEAT distance, and this is quite a compelling modification to add to any CoDeepNEAT implementation.

4.1.3 Attribute distance

The inclusion of an attribute distance does allow CoDeepNEAT to compare individuals in a more fine

grained manner, which may explain why the three attribute distance runs in figure 8 are so similar after generation 10. Even though Meyer-Lee G. et al. [4] did not provide any comparison between attribute distance and base CoDeepNEAT, they still achieved good results when using attribute distance. Our results (figure 8) seem to contradict this, since attribute distance seems to make accuracy strictly worse. However our method of performing attribute distance was different to the method used by Meyer-Lee G. et al. [4] and does require extensive tuning. These factors are likely the cause of the poor results.

4.1.4 Representatives

Figure 9 shows that REPCDN (section 3.2.4) is strictly worse than SPCDN (section 3.2.1). At the same time it is the most volatile (inside of a single run) of all the versions of CoDeepNEAT that were tested, as it seems to have many, seemingly random, drops in accuracy. This volatility and overall low performance are, at least in part, a product of a lack of tuning, due to the many tunable parameters REPCDN introduces (table 2).

4.1.5 Comparison to original CoDeepNEAT

It is important to note that our implementation of CoDeepNEAT did not achieve as high an accuracy as the original implementation. This is due to five factors. First, we trained ANNs for only five epochs while in the original paper ANNs were trained for eight epochs [2]. Second, original CoDeepNEAT used data augmentations while no data augmentations were used in our implementation. Third, we trained up to a maximum of 30 generations while the original paper [2] trained for 72 generations. Fourth, the CoDeepNEAT authors likely had time to tune their hyper-parameters, thus squeezing more performance from the algorithm. Fifth, our implementation of the aggregation of skip connections is naive and likely causes a performance bottleneck. All of these factors contribute to the worse performance of our implementation, especially when considering the poor improvement in accuracy seen when fully training (table 1). Even though our implementation did not perform as well as original CoDeepNEAT, any modification that improved our original CoDeepNEAT implementation would likely improve any other, better tuned, implementation.

Taking all of the above into account elite CDN (figure 5) did achieve a higher accuracy than the original CoDeepNEAT [2] inside of a the evolution, that is before being fully trained. Elite CDN achieved a highest accuracy of 82.81% when only training for 5 epochs and evolved for 30 generations, while original CoDeepNEAT does not explicitly state their accuracy during evolution, they do mention they achieved 80% accuracy in 12 epochs during a fully training run [2]. However when fully training elite CDN only achieved 86.15%, while original CoDeepNEAT achieved 92.3% [2]. This shows that our implementation has some bias towards training ANNs faster and not longer.

To push for better results we ran a single 20 epoch experiment on elite DACDN with node breeding which is a version of CoDeepNEAT which co-evolves data augmentations [8] and inherits interpolated values from parents modules nodes when crossing over individuals [7]. This achieved 87.69% accuracy during evolution and 92.12% accuracy while fully training, showing that longer epochs during evolution increase an ANNs ability to train for longer. When comparing this to the 80.6% accuracy of our original CoDeepNEAT implementation, it shows the benefit of combining all proposed modifications and increasing the epoch count, to such a degree that it improves the overall accuracy by 11.52%.

4.1.6 Computational Time

Our second best and second most expensive run took 48 hours and achieved an accuracy of 91.11% using 4 Nvidia Tesla v100[4] to run the experiment. Compared to AmoebaNET [6], which is another NE approach that achieved 96.6% accuracy on CIFAR-10. To achieve this, they used 450 Nvidia K40[5] GPUs and 7 days, this equates to around 390 times more GPU hours than our implementation of CoDeepNEAT. It must be noted that the Tesla v100 is significantly faster than the K40, however this does not fully account for the massive difference in GPU hours between the two algorithms. The results show that CoDeepNEAT is less performant, but is conservatively an order of magnitude faster than AmoebaNet.

4.1.7 Summary

Both attribute distance and REPCDN performed poorly, which is in direct contradiction of the hypothesis (section 1.1). They failed to stabilize the evolutionary process and thus did not increase the overall accuracy of base CoDeepNEAT. However, this could be improved by tuning the hyperparameters in future work.

The positive results that SPCDN and SPCDN with GED produced prove our hypothesis correct. These modifications greatly increased the stability of CoDeepNEAT even when combined with other modifications. This shows how important it is for the functional niche of a species to remain constant. Therefore we recommend any implementation of CoDeepNEAT to use the speciation strategy outlined in section 3.2.1 along with replacing the NEAT distance function (formula 1) with graph edit distance.

It should be noted that our implementation of CoDeepNEAT is significantly faster than AmoebaNet [6] and even though it is less performant, it provides an option for those who do not have the resources needed for a NE algorithm as computationally expensive as AmoebaNet.

5. Future work

There are a lot of optimizations that could be done to our implementation of CoDeepNEAT in terms hyper-parameters tuning. This would likely squeeze

[4] https://www.nvidia.com/en-us/data-center/tesla-v100/
[5] Newer version of the k40:
https://www.nvidia.com/en-gb/data-center/tesla-k80/

some extra performance, however we did not have time to perform this.

Our implementation of CoDeepNEAT should be tested on more datasets and different fields (such as natural language processing) to verify its fidelity. If tested on larger datasets, such as ImageNet, GED would likely be highly beneficial, due to its ability to better distinguish complex graphs, which are necessary for datasets as large as ImageNet.

The biggest room for future work lies in trying to find out why fully training certain graphs only increases the maximum evolved accuracy by around 5% (table 1). We did see an improvement in accuracy gains when training for more epochs, however this does not compare to accuracy gains seen in original CoDeepNEAT (table 1).

6. Conclusion

The fact that SPCDN (section 3.2.1) improved the results of both modmax (section 3.3.2) and base CoDeepNEAT show that it is certainly a worthy inclusion in any implementation of CoDeepNEAT. Along with this GED (section 3.2.2) showed that it helps improve on SPCDN and it removes some hyper-parameters, which is highly beneficial to any autoML algorithm. On the other hand both attribute distance (section 3.2.3) and REPCDN (section 3.2.4) performed poorly and likely would need extensive tuning in order for their performance to improve.

The 11.52% accuracy improvement that our best version of CoDeepNEAT (table 1) makes over our base implementation, shows the benefit of combining all of the best proposed modifications in this paper and in [7, 8]. Thus it is highly recommended to use this version as it would likely perform even better on a well tuned version of CoDeepNEAT.

The time taken by our implementation of CoDeepNEAT to evolve a suitable model for the CIFAR-10 dataset is much faster and uses less resources than other NE methods [6]. The results show that it is possible to achieve satisfactory results using our CoDeepNEAT implementation, which does not require excessive amounts of computational power and time. This provides an alternative for people who have not been able to use popular NE methods, because of their extreme computational cost.

Acknowledgements

Thanks to Geoff Nitschke for the guidance and corrections when writing the report. Thanks to Shane Acton and Liron Toledo for help implementing CoDeepNEAT and many ideas exchanged on how to modify it.

References

[1] Liang, J., Meyerson, E., Hodjat, B., Fink, D., Mutch, K. and Miikkulainen, R., 2019. Evolutionary Neural AutoML for Deep Learning. arXiv preprint arXiv:1902.06827.

[2] 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.

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

[4] Meyer-Lee, G., Uppili, H. and Zhuolun Zhao A. 2018. Evolving Deep Neural Networks. [online] Retrieved August 28 2019 from https://github.com/zhaolebor/evolving-deep-neural-networks

[5] 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.

[6] 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).

[7] Acton, S. 2019. 'Improving CoDeepNEAT with elitism and stability'. Honours. University of Cape Town. Cape Town

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

[9] Feurer, M. and Hutter, F., 2019. Hyperparameter optimization. In Automated Machine Learning (pp. 3-33). Springer, Cham.

[10] Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M. and Hutter, F., 2015. Efficient and robust automated machine learning. In Advances in neural information processing systems (pp. 2962-2970).

[11] Zoph, B. and Le, Q.V., 2016. Neural architecture search with reinforcement learning. arXiv preprint arXiv:1611.01578.

[12] Floreano, D., Dürr, P. and Mattiussi, C., 2008. Neuroevolution: from architectures to learning. Evolutionary intelligence, 1(1), pp.47-62.

[13] Abbass, H.A., Sarker, R. and Newton, C., 2001. 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.

[14] Deb, K., Agrawal, S., Pratap, A. and Meyarivan, T., 2000, September. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In International conference on parallel

problem solving from nature (pp. 849-858). Springer, Berlin, Heidelberg.

[15] 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.

[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

[20] 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.

[18] 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.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity Mappings in Deep Residual Networks. CoRR abs/1603.05027 (2016). http://arxiv.org/abs/1603.05027

[20] Gao, X., Xiao, B., Tao, D. and Li, X., 2010. A survey of graph edit distance. Pattern Analysis and applications, 13(1), pp.113-129.

Appendix A

The tables in this section describe the parameters we used for the experiments to allow for replication of our results.

Evolution parameter	Value
Module population size	56
Module node mutation chance	8%
Module connection mutation chance	8%
Target number of module species	4
Blueprint population size	22
Blueprint node mutation chance	16%
Blueprint connection mutation chance	12%
Species number mutation chance (early)	50%
Species number mutation chance (late)	13%
Percent of individuals that crossover	20%
Percent of elite individuals that survive to the next generation	10%
Total number of ANNs created and evaluated in a generation	75
Layer size mutation chance	22%
Layer type mutation chance	8%
Disjoint coefficient (NEAT distance)	3
Excess coefficient (NEAT distance)	5
Layer size coefficient (attribute distance)	2
Layer type coefficient (attribute distance)	2
Closest representatives to consider - REPCDN (section 3.2.4)	6
Representative mutation chance - REPCDN (early)	60%
Representative mutation chance - REPCDN (late)	20%
Chance to mutate all of the same representatives in a genome in the same way - REPCDN	20%

Table 2: Hyperparameters used in the evolutionary process of CoDeepNEAT

Experiment parameters	Value
Dataset	CIFAR-10
Epochs	5
GPU	Tesla v100
Number of GPUs	2-4
Repeating tests	2-6
Data augmentations	None

Table 3: Hyperparameters used for the experiment setup of CoDeepNEAT