Improving CoDeepNEAT’s performance using co-evolved data augmentations

Liron Toledo
University of Cape Town
Cape Town, South Africa
liron.toledo@outlook.com

ABSTRACT

Data Augmentation is a regularisation technique used in machine learning that artificially inflates training setsin order to better train artificial neural networks. However, it is often difficult to determine the best set of augmentations to use to maximise performance.. The goal of this paper is to use CoDeepNEAT, which is a neuro-evolution algorithm used to evolve network topologies to automatically find the best data augmentation scheme for a network architecture and dataset. To do this we propose DACDN and Elite DACDN which are extensions to CoDeepNEAT that co-evolve data augmentation schemes alongside the networks themselves. The schemes that were produced by DACDN and in particular elite DACDN managed to considerably increase the accuracy of the networks that they were evolved with. Despite not achieving state-of-the-art results, elite DACDN showcases the potential of a coevolutionary approach to automated data augmentation, not only as a viable extension to CoDeepNEAT, but within the field of neuroevolution as a whole.

1 INTRODUCTION

Machine learning is increasingly being used in order to solve modern problems [20, 21]. In particular, deep neural networks (DNN) have proven to be extremely successful in achieving state-of-the-art results in numerous fields [29, 30]. The only caveat being that DNNs require long training times and large amounts of high quality training data to achieve desired results. A DNN that fails to be provided these two requirements will never achieve satisfactory performance [8].

As the applications of deep learning algorithms continue to grow [21], there exists an increasing demand for useful labeled training data. Unfortunately, finding large amounts of labeled training data for certain tasks can be extremely difficult [22, 23]. This is the core problem that this paper hopes to address.

In order to combat scenarios where there exists a severe lack of training data, a technique known as data augmentation (DA) is used. DA enlarges a dataset by performing transformations that preserve the data’s relevant labels. For instance, in the context of image data, one might rotate the image of a cat to create a new image but that new image will still resemble a cat [3, 4].

The selection as well as the order of application of various DAs create what is known as the DA scheme. There are countless different schemes that can be created and in order to make full use of DAs it is paramount that a good scheme is formulated.

Researchers have found that certain DA schemes, when used on the same training data, create datasets that train DNNs better than others [1, 2]. With this in mind, it is important to utilise a DA scheme that maximises the accuracy increase. This process, if done manually, could take many testing iterations to achieve satisfactory results.

This paper attempts to use the neuro-evolution (NE) [5] algorithm known as CoDeepNEAT (CDN) [1, 2] in order to automate the DA scheme selection process. CDN has shown to be extremely effective at finding DNNs that outperform both handcrafted networks [1, 2]. Additionally, Because CDN is an automated machine learning (AutoML) [6] algorithm, it requires very little domain knowledge or human intervention to use effectively.

We built a DA based extension to CDN that, given a specific dataset, will find the optimal DA scheme from a pool of generic DAs (see appendix B). We chose to use DAs because of their history of improving the performance of small and noisy data-sets (see section 2.4).

1.1 Research Question

Determine if one can adapt CDN, to create an automated system that finds a DA scheme, which maximises the performance increase for a given network architecture (see section 2.3.2) when trained on a particular data set (see section 4.4).

2 BACKGROUND INFORMATION

2.1 Deep Learning

Deep Learning is an approach within the larger field of machine learning [8]. It refers to the creation of artificial neural networks (ANNs) [7] with multiple hidden layers which are referred to as deep neural networks (DNNs) [8]. Deep learning is used extensively to tackle larger and more complex problems that simple 3 layer ANNs cannot [8]. This can be seen in fields such as computer vision [9], which often use deep learning architectures such as Convolutional Neural Network (CNN) [10] for various tasks such as image classification, image segmentation or object detection.

However, powerful performance such as that found in CNNs come at a price. Large networks with many layers and connections are extremely computationally expensive to train and certain models can sometimes take days to fully train, even with the use of powerful hardware [8, 10].

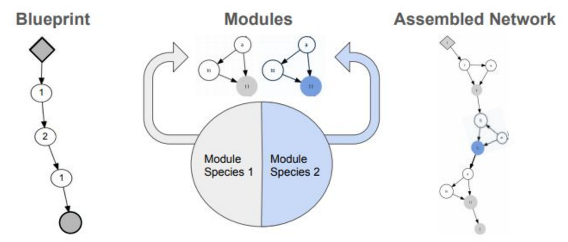


Figure 1: Depiction of the process of converting blueprints into assembled networks, by sampling from module species [1]

However, powerful performance such as that found in CNNs come at a price. Large networks with many layers and connections are extremely computationally expensive to train and certain models can sometimes take days to fully train, even with the use of powerful hardware [8, 10].

With all this in mind, it is easy to see how important both the acquisition of high quality data and construction of suitable network topologies can be. Fortunately, both of these problems can be addressed through the use of optimisation methods such as Evolutionary Algorithms [11] and regularisation techniques such as data augmentation [3, 4].

2.2 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are a group of algorithms inspired by biological evolution for optimisation purposes. EAs use the principle of Darwinian evolution to find the fittests members (individuals) in a population. This is done using a predefined fitness function which measures how effective an individual is at a specific task. Fit individuals are selected for crossover (analogous to sexual reproduction) and mutated to create new generations of individuals. This process is then repeated for a specified number of generations. Finally, the fittest individual is chosen from the final generation.

2.3 Neuroevolution

Neuro-evolution (NE) is a subset of A.I that uses evolutionary algorithms to evolve networks by manipulating their hyper-parameters, topology or rules [5]. Initially, NE was used only to evolve hyper-parameters [17, 18]. Later iterations, namely the topology and weight evolving neural networks (TWEANNs) NE algorithms search for optimal topologies and weights for a network [17, 18].

2.3.1 Neuro-Evolution of Augmenting Topologies

NEAT is a member of the previously mentioned TWEANN group of NE algorithms [17, 18, 19]. Meaning that it attempts to find a network's optimal weights and its best topology. NEAT achieves this by evolving the network's nodes (or neurons) in addition to the connections between these nodes. Furthermore, it attempts to optimise the weights of the network's connections on each iteration of the topology.

NEAT does not suffer from a common problem that exists in many other traditional NE methods called competing conventions [17].

Simply put, this problem occurs when two networks that have a similar fitness but radically different topologies crossover. The children (mixed topologies) of these crossovers often perform worse than either of their parents because they are unlikely to take the best aspects of two networks that are very topologically different.

NEAT solves this problem through the use of speciation but mostly through the application of historical markings [17]. Speciation involves clustering the population by their similarity to a given species' representative. Every generation a random individual is chosen from every species to be their species' representative. Historical markings allow individuals to keep track of their lineage. NEAT only crosses over networks within the same species, in other words, only networks with the same heritage (overlapping parents) will be crossed over. This minimizes the cases where crossover results in damaged offspring [1] thus maximizing efficiency.

In order to protect innovation, individuals are only allowed to compete with members within its own species. Species that perform better will grow larger than those that perform poorly. NEAT then proceeds to evaluate every individual (network) and assign each a fitness score. Those with the highest fitness scores may be crossed over and mutated. However, if an individual X crosses over with an individual Y, their child will inherit both the common connections of the two parents as well as the connections that only belong to the fittest parent. Mutations involve either changing the topology (adding nodes and connections) or the weights of the ANN. All new children created will be grouped into their respective species as the next generation.

2.3.2 CoDeepNEAT

CoDeepNEAT (CDN) is a recently developed co-evolutionary, multi-objective NE algorithm which has achieved state-of-the-art results [1]. As the name implies CDN, is an extension of NEAT, however it is significantly different in a number of ways.

In order to better understand how CoDeepNEAT works, it would be best to first explain a somewhat intermediary step between NEAT and CoDeepNEAT known as DeepNEAT [1]. DeepNEAT extends NEAT but with two notable changes. The first being that DeepNeat is not a TWEANN algorithm. Unlike NEAT, DeepNeat is only concerned with the hyper-parameters and topology of individuals and does not attempt to optimise weights. Weights are instead updated during backpropagation [12]. The second difference being that unlike in NEAT, where every node in an individual represents a single neuron, in deepNeat, every node now represents a different deep layer in the DNN. This abstracted representation of a DNN is known as the genotype. The connections between nodes in DeepNeat are not weighted as they only indicate how layers pass their outputs to each other. Nodes ordinarily represent convolutional or fully-connected layers.

The primary difference between CoDeepNEAT and DeepNEAT is the use of coevolution. Coevolution takes place between two populations, namely, blueprints and modules. [1]. Modules represent small deep neural networks. Meaning that similar to DeepNEAT, every node in the graph represents a layer in the

ANN and hold all the required information needed to convert itself to a neural network layer (ie. activation function, kernel size, layer type, layer size). Blueprints take the form of graphs where every node points to a specific species of modules. CoDeepNeat constructs large and complex ANNs by combining modules.

A phenotype (DNN) is constructed by a group of modules which are joined together by ablueprint. . The Blueprint is used to construct a graph of connected modules by transforming every node in its graph into a randomly selected module from the species that the node is pointing to. However, once a node has randomly chosen a module from a species, every other node that points to the same species will choose the same module previously chosen from that species. Meaning that no blueprint can have more than one unique module from the same species. This is where the repeating structures comes from.

In CoDeepNEAT, both the modules and blueprints are evolved separately using the NEAT algorithm. This is where the cooperative evolution component comes into play. The graphs created by blueprints are essentially larger more complex graphs assembled by fusing together many different smaller networks (created from modules). The graph of modules created by the blueprint can then be parsed into an DNN phenotype and trained for a limited amount of time on some training data set. After training, the accuracy of the network on a testing set determines the fitness of all the modules used in the network as well as the blueprint itself. The fitness value is then used as a basis for decision making regarding the mutation and crossover for all the modules and the blueprint. CoDeepNeat will ideally find some top performing network which will then be taken and trained such that it converges correctly on unseen data sets.

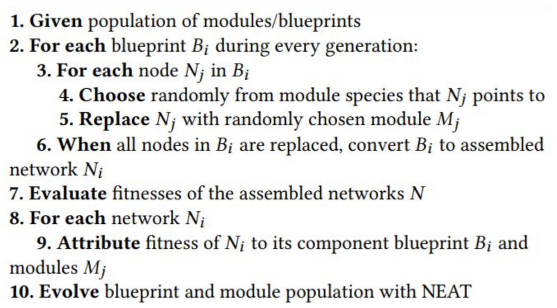
2.4 Data Augmentation

Data Augmentation is a regularisation technique that artificially inflates the training data-set by performing label-preserving transformations to add more uniform examples [3, 4].

Regularisation, simply put, attempts to restrict the network at training time. Regularisation methods attempt to limit the complexity of a network either in the network's weights or in the data that it uses. In doing so it addresses the two major causes for overfitting, a lack of quality labeled training data and a network architecture that is too complex [13]. The end goal being to eliminate overfitting and illicit faster generalisation.

DA has proven to be excellent in reducing the overfitting in ANNs caused by training on too little data [3, 4, 14]. A DAs scheme is what determines how your DAs are applied. There exists countless different schemes that can be created, this can make finding a good DA scheme a very difficult task. Some examples of DA schemes include a scheme that rotates 50% of images and flips the rest or a scheme that converts all images to grayscale and then translates them, the possibilities are endless.

As mentioned prior, deep learning algorithms such as CNNs require huge amounts of training data to perform well [8]. Unfortunately, it is not always possible to find large amounts of quality labeled data for a given problem [22, 23]. As a consequence, it can be difficult to implement deep learning



```
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].

algorithms for certain tasks. However, it is possible to use DA to inflate small datasets such that we can get satisfactory performance out of a variety of deep learning algorithms [14, 22, 23]. Nevertheless, one must still be careful in choosing appropriate DAs for their respective datasets. For instance, one can easily see how vertically flipping images of text and numbers would result in images that are no longer label preserving. The type of data augmentation used in this paper is known as generic DA.

2.4.1 Generic Data Augmentations

Generic augmentations consist of computationally inexpensive transformations that alter the geometry of an image (Geometric transformations) or the light or colour of an image (Photometric transformations). Some examples of generic DAs include: rotating, cropping or colour jittering (Manipulation of HSV) [3].

Generic DAs can be combined with each other to create more complex but more powerful augmentations. For instance, one might combine rotating and cropping. However, despite the fact that one can create more powerful DAs through the combination process [4, 15, 16], one also runs the risk of either creating an augmentation that is no longer computationally inexpensive or even worse creating a DA that worsens performance by combining augmentations that don't lend themselves to each other well (e.g. greyscale and colour jittering).

2.4.2 Automated Data Augmentation

It has also been found that certain data augmentations result in inflated data sets that better generalise and increase the accuracy of networks when used on the same data set [3, 4]. As such, we can infer that certain augmentations work better than others for different network architectures and datasets. In light of this, researchers have attempted to produce what is referred to as automated DA systems.

Automated DA systems use some algorithm to automatically find some optimal DA scheme that will result in the best possible performance increase for a specific network architecture and data set and do so with minimal human involvement or interference. Notable examples of automated DA systems include: Neural augmentation [4], Smart Augmentation [15] and AutoAugment [16]

3 Methods

We based our implementation of CoDeepNEAT (CDN) on what was found in [1, 2]. All code and experiment data for our implementation will be available on GitHub[1].

3.1 CoDeepNEAT Re-implementation

During our construction of CoDeepNEAT we rarely encountered scenarios where certain implementation details were not fully disclosed in [1, 2]. When such situations arose, we were forced to implement those sections based on our own ideas as to how the algorithm should operate. Additionally, there were also times where we chose to slightly deviate from the algorithm, as outlined in [1, 2] and implement our own very minor changes.

Two significant aspects not specified in [1, 2] are how CoDeepNEAT chooses its starting population's topology and its strategy regarding speciation. We chose to use two seperate starting topologies for both modules and blueprints. Modules and blueprints can both begin with either a three node triangular structure or a linear topology consisting of one input node and one output node

In Liang, J et al [1], the number of module species that the algorithm aims for was four but the method they used to achieve this was not specified. Our implementation chose to use a speciation strategy that also aims for four module species by increasing and decreasing the speciation threshold. The speciation threshold determines whether or not an individual is added to a species. This is done by checking if an individual's similarity to the species' representative is less than the threshold, in which case it is added to the species.

During testing we observed that the total number of species created in the first generation was usually only two despite our target being four. We concluded that this was because the initial population of blueprints and modules only consisted of two different topologies (only two groups of similar individuals exist). Because blueprint nodes can only choose from the species available to them, having so few species to choose from initially is an obvious minor issue we could not overlook. We solved the problem by making sure each blueprint node had a high probability to change their species until the target number of species was reached (four). In doing so, species linked by blueprint nodes received a better overall distribution early on and all modules were allowed to have a better chance at being evaluated.

3.2 CoDeepNEAT extensions

After implementing CoDeepNEAT, there were a number of significant extensions and modifications made in order to either fix inherent large-scale flaws with the algorithm or to improve the algorithm's overall performance.

3.2.1 Module Retention

We found that one of CoDeepNEAT's biggest flaws was in regards to its elitism. It proved to be extremely inept at holding on to the best performing DNNs(phenotypes)in a generation. This is because CoDeepNEAT randomly assembles DNNs using different modules from the same species each generation.

The module retention extension was built to address this problem by ensuring that elite blueprints hold onto their modules whenever possible. Ideally, this extension should prevent performant blueprints from randomly selecting modules each generation and instead retain the modules they used in the last generation. This should help prevent each generation's maximum accuracy from dropping often.

However, not all modules are necessarily retained by elite blueprints. Blueprints will only hold on to modules if they are also in the elite (perform well). This means elite blueprints often lose some of their modules, requiring them to be resampled, despite this they remain similar enough to still improve CoDeepNEAT's overall elitism.

3.2.2 Maximum Aggregation

In traditional CoDeepNEAT, in the event that a single module or blueprint is evaluated multiple times, that same individual will receive the average fitness of all the evaluations. In order to further improve module retention, we instead made CoDeepNEAT choose the maximum fitness of all the evaluations. This ensures that the very best blueprint as well as all of its modules live to the next generation, as a blueprint being elite will guarantee that its modules are elite too.

3.2.3 Speciation

The speciation extension's primary goal is to keep species more stable, meaning the function they represent does not change rapidly over time. More stable species not only makes the functional niche that a blueprint points to less random but also greatly benefits the previously mentioned module retention extension.

In NEAT, individuals are placed in species based on a species representative. An individual will be placed in the first species whose representative is similar enough to the individual. This method presents two problems:

1. Species who are situated closer to the beginning of the species list have an unfair advantage over other species.
2. An individual might be put into a species that appears early on when there might be a better suited species later in the list. This could lead to the creation of different species that represent very similar individuals.

The first change that the species extension makes is how individuals are grouped into species. Instead of depending on the species list for grouping, individuals are placed into a species if they are most similar to a species' representative (regardless of the species position in the list). This should not only prevent the algorithm from biasing certain species, but also help give species a more meaningful representation.

In NEAT, the method whereby species representatives are selected is rather simple. Every generation, a random individual is chosen from every species to be that species' representative. This

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

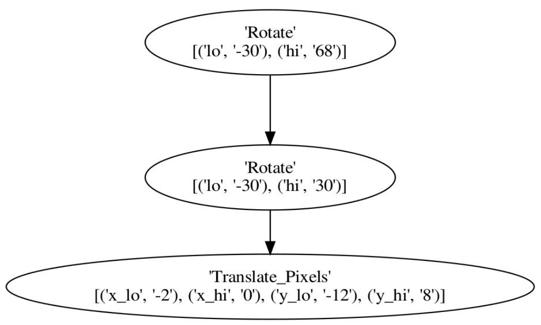

'Rotate'
[('lo', '-30'), ('hi', '68')]
'Rotate'
[('lo', '-30'), ('hi', '30')]
'Translate_Pixels'
[('x_lo', '-2'), ('x_hi', '0'), ('y_lo', '-12'), ('y_hi', '8')]

Figure 2: Example of a DA genotype, which is used in the creation of a DA scheme

is problematic because if an outlying representative is chosen it could cause a species to be poorly represented.

The second change that the species extension makes revolves around improving representative selection. Instead of using random selection, the individual that is most similar to each member in its species is chosen as the representative. By measuring the similarity of each species member in relation to every other species member we can find the desired representative by selecting the member with the smallest summed similarity to all other members. Similarity is quantified using the method outlined in [28]. Although this modification adds additional computation complexity, this complexity is negligible in comparison to DNN training. Ideally, this extension should allow for representatives to more accurately represent their species from generation to generation.

For additional information on the implementation of module retention and speciation extensions refer to [26] and [27] respectively.

3.3 Data Augmentation & Neuro-Evolution

Normally, networks evolved by neuro-evolution algorithms can be trained after evolution with a DA scheme in order to boost their performance. However, as mentioned earlier in the paper, different DA schemes perform better on different network topologies and datasets [3, 4]. Finding the optimal DA scheme manually, if at all, can take a long time.

In [1], a number of DAs were used as global hyperparameters. The parameters of those DAs were chosen to be evolved in order to boost the performance of their networks. However, the DAs themselves were manually picked and could only ever be applied in the same static order. Most importantly, unlike our co-evolutionary approach, they attached their DA parameters to blueprints and evolved them separately. This means that a good DA scheme linked to a bad blueprint would be lost. However, when evolving DAs in a separate population, evolutionary progress made by DA genomes is more robust, as the same DA genome may be linked to many blueprints, allowing for a chance for good DAs to survive being used by bad blueprints .

```
1. Given population of modules, blueprints and data augmentations
2. Create N data augmentation scheme graphs consisting of two nodes with each node representing a random augmentation with X parameter values
3. For each blueprint $B_i$, modules $M_j$ and data augmentation scheme $DA_k$
    a. Assign $DA_k$ to a $B_i$
    b. Convert $B_i$ to assembled network using $M_j$ and convert $DA_i$ to practical DA scheme
4. Evaluate fitnesses of all assembled networks on the dataset created by assigned practical DA scheme
5. For each assembled network $N_i$
    a. Attribute fitness of $N_i$ to its component blueprint $B_i$ and modules $M_j$ and data augmentation scheme $DA_k$
6. Evolve blueprint, module and data augmentation populations with NEAT.
```

Algorithm 2: Evolution and training procedure followed by DACDN

4 CODEEPNEAT FOR DATA AUGMENTATION

One of CoDeepNEAT's biggest strengths is its capability for coevolution. Traditionally, CoDeepNEAT co-evolves two populations (blueprints and modules) but this paper attempts to add a third population for co-evolution, namely, data augmentation schemes.

The DA extension for CoDeepNEAT, which will henceforth be referred to as DACDN uses an evolutionary strategy to find the optimal DA scheme for a network architecture and dataset. In doing so, DACDN can maximise the performance increase gained when using DAs, without the difficulty of finding a quality DA scheme manually. It does this by co-evolving a population of data augmentation schemes (and their parametres) alongside blueprints and modules. This means that not only were the DA schemes themselves evolved but the parameters of each individual DA was evolved at the same time as well. Blueprints link to DA individuals, and pass their DA individual links down to their children. Additionally, when blueprints mutate, they also have a chance to resample a new DA individual.

4.1 Data Augmentation Population

The DAs chosen for evolution (see appendix B) were selected from DAs that were shown to perform well in the past [24, 25] in addition to DAs that we hypothesised would illicit good results.

An individual in the DA population is constructed as a linear directed graph as seen in figure 2. Every node in the graph represents an individual DA, while the entire graph (genotype) represents the DA scheme. It is important to note that DA graphs are directed, this means that there is an order to which each DA in the graph is applied to an image. This is important as certain DAs can produce schemes that augment images differently based on the order they are applied.

Each individual in the DA population has their own set of evolvable parametres (see appendix B). These parametres change how an augmentation is applied to an image (e.g. more or less extreme rotation). DAs used by DACDN often use a non-deterministic approach for their parameter values. In other words, parameters often take the form of ranges with an evolvable upper and lower bound. Every time a DA is used, a random value within that range is chosen as its actual parameter value.

The chosen population of DAs is very crucial to the success of DACDN. If the population is populated with non-performant or badly initialised DAs, DACDNs performance will be severely negatively affected. Additionally, one must take into consideration the dataset being used. Certain DAs will not illicit good results for certain datasets, for instance using a flip DA with a dataset consisting of text such as MNIST[2] could produce non-label preserving images, which is clearly not desirable.

4.2 Co-Evolving Data Augmentation Schemes

DACDN co-evolves DA schemes by assigning a DA scheme to a blueprint in the population. As a blueprint evolves its topology, DACDN evolves its associated DA in three possible ways: it can tweak an individual DA node's parameters, add a completely new node to DA scheme or remove existing nodes from the DA scheme.

In CDN a blueprint's fitness is returned as the classifying accuracy of the DNN it creates. Normally, a blueprint's assembled DNN is trained batch by batch for a specified number of epochs. However in DACDN, it is first trained for one epoch on the original training data and then trained for an epoch on the augmented data. To summarise, DACDN trains networks on double the data meaning double the specified epochs. The fitness that the blueprint receives by training on the in DACDN is also used by the DA scheme for its own evolution purposes.

DACDN allows blueprints to not only exploit a population of modules but also a population of DA schemes. The population of blueprints should evolve such that they find both the optimal topology and DA scheme to maximise fitness. Refer to Algorithm 2 for further details on how DACDN operates.

One possible DA that DACDN has access to is known as the No_Operation augmentation. No_Operation, as its name implies, performs no augmentation on its input data. By including it, we have given DACDN the option to not augment its data or replace an existing DA scheme node with No_Operation (effectively removing that node). In the worst case DACDN would simply double its data with No_Operation but ideally we will see that it chooses to use the DAs provided instead.

Originally we planned to initialise every blueprint with either a single DA or a No_Operation. Instead, we chose to initialise all starting blueprints with a DA scheme consisting of two random nodes (e.g. Rotate → Translate). This was done to both speed up the DA scheme evolution process and to improve accuracies of blueprints earlier on.

In CoDeepNEAT, blueprints are parsed multiple times into different DNNs, each time they are parsed they pick different modules from the species their nodes point to. This is done to better explore the module population. Following on from this idea, in DACDN, each time a blueprint is parsed, there is a 20% chance that the blueprint ignores its current DA scheme in favour of trying a completely new one. If the new DA scheme performs better, then it replaces the old scheme in the blueprint genome. There is an extremely small chance that all parsings ignore their current DA but this risk is outweighed by the benefit of better exploration of the DA population.

4.3 Limitations

DACDN is a powerful extension but it is limited in some regards. Particularly in its training time and its dependence on the DA population.

4.3.1 Training Time

DACDN introduces the added overhead of having to augment the entire dataset and often with more than one data augmentation. Additionally, DACDN makes the blueprint's assembled DNNs train on double the data (double the epochs). The combination of these two factors result in DACDN taking approximately double the time that regular CDN takes to complete a specified number of generations.

4.3.2 Dependance on data augmentation population

DACDN is entirely dependent on the population of DAs it has access to. If it is initialised with a list of non-performant or badly suited DAs it is likely that performance increase will be minimal if at all. Additionally even if good DAs are used if their parametres are initialised poorly, it could make DACDN take far longer to find good individuals.

4.4 Experimentation

Experimentation was performed using four separate configurations of our implementation of CDN. These are Base CDN, Elite CDN, DACDN and Elite DACDN.

Base CDN refers to our CDN implementation based on information found in [1, 2]. Elite CDN refers to CDN with the module retention, max fitness aggregation, and speciation extensions being used simultaneously. DACDN refers to base CDN with the DA extension. Finally, Elite DACDN refers to Elite CDN with the DA extension.

The primary goal of all experimentation was to showcase the performance difference between CDN and DACDN both with and without specific extensions [see section 3.2]. All experimentation was performed with Nvidia Tesla v100 GPUs and implemented parallelism techniques (see appendix A) in order to speed up experimentation.

All tests were benchmarked using CIFAR-10[3]. CIFAR-10 consists of 50,000 training images and 10,000 testing images. Images consist of 32x32 color pixels and belong to one of 10 classes. CIFAR-10 was chosen as the primary benchmarking dataset so that we were able to compare our implementation of CDN to that found in [1, 2], which also used CIFAR-10.

[2] http://yann.lecun.com/exdb/mnist/

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

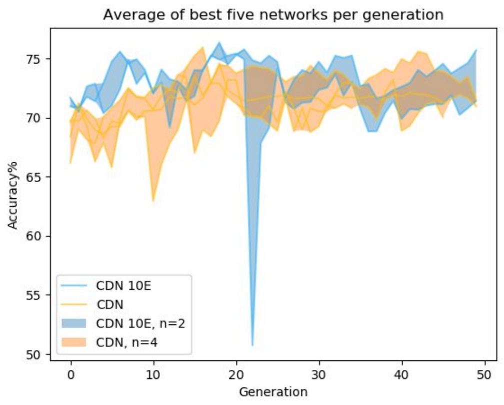


Figure 3: CDN trained with 5 epochs compared to CDN trained with 10 epochs.

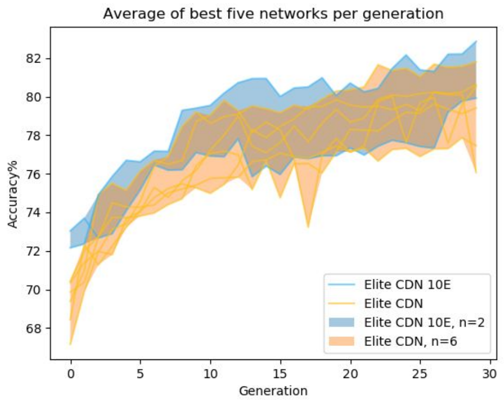


Figure 4: Elite CDN trained with 5 epochs compared to Elite CDN trained with 10 epochs

In order to ensure consistency, DACDN and Elite DACDN experiments were all ran for 50 generations and 5 epochs. Experiment were benchmarked against Base CDN and Elite CDN. As mentioned prior, DACDN uses effectively double the epochs it is given. This is because networks train for the specified number of epochs on both the original data and the augmented data. Knowing this, in order to allow for a fair comparison, non-DA experiments were also ran with 10 epochs.

Because the original CDN [1] implementation that experimented with CIFAR-10 was single-objective, all experiments were performed using a single-objective version of both CDN and DACDN. This is so we may perform a fair comparison of our CDN implementation to theirs.

We strived to perform at least two runs with identical parameters were for every test in order to record and control for the consistency (or lack thereof) between them. We wanted to ensure that the discrepancy between separate runs were at least somewhat visible in our results despite the limited time we had for experimentation.

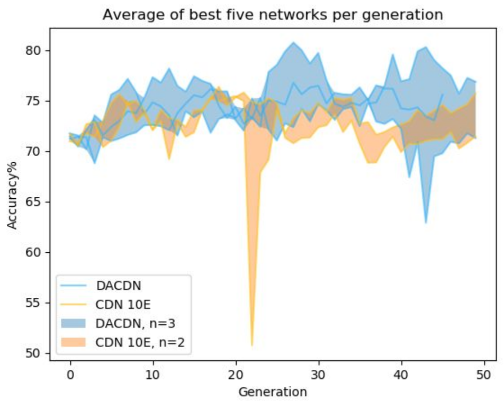


Figure 5: DACDN compared to CDN

6 RESULTS AND DISCUSSION

The following section outlined the results we found during experimentation as well as any notable conclusions we could draw from them. Comparisons will be aided by figures that show the average accuracy of the top five networks for each generation. Additionally, figures highlight the boundaries of multiple runs in order to help better distinguish the range of performance a CDN variation achieved.

6.1 Base CDN & Elite CDN

In figure 3 and 4 we see both base CDN and elite CDN when given additional training time manage to increase their performance however this performance increase was not significant.

Once again, it is important to note that DACDN and elite DACDN train their networks for effectively double the amount of epochs than specified (5 epochs). In order to produce a fair comparison, DACDN and elite DACDN will henceforth be juxtaposed against the 10 epoch runs of both CDN and elite CDN.

6.2 Base CoDeepNEAT & DACDN

The following results belong to experiments involving the base implementation of CoDeepNEAT and DACDN and are detailed in figure 5. To begin, we would first like to illustrate the performance increase DACDN achieved in comparison to base CDN.

DACDN is seen to produce networks that are at worst as performant as those produced by CDN 10E, but at best vastly superior. You can see that DACDN even manages to evolve networks with around 80% accuracy. This is a feat that our implementation of CDN never manages to come close to.

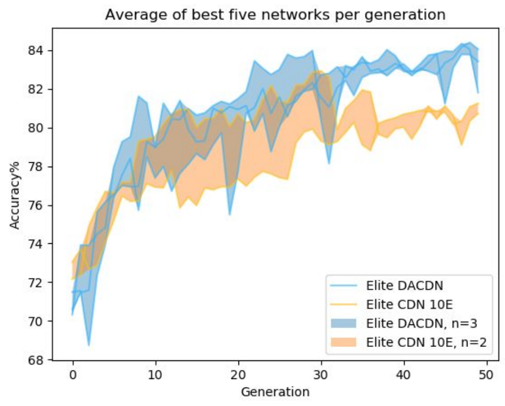


Figure 6: Elite DACDN compared to Elite CDN

Unfortunately DACDN still faces many of the same problems that CDN does. The most important one being its lack of elitism and subsequent randomness. We see in figure 5 that CDN, regardless of its training time, constantly experiences steep drops in performance that it has to recover from. We hypothesise this is caused by base CDN's inability to effectively hold on to its best performing individuals. This shortcoming can cause different runs to be vastly differing in performance. The same can be said about DACDN but with an additional component to consider. DACDN has to manage the entire DA population as well as blueprints and modules. This leads to two additional scenarios in which a drop in performance might occur. One, a co-evolved elite DA scheme might be lost in favour of one that performs worse. Two, a mutation to a DA scheme might result in a bad scheme which is no longer label-preserving.

6.2 Elite CoDeepNEAT & Elite DACDN

Figure 6 illustrates the performance increase Elite DACDN achieved in comparison to Elite CDN. Elite DACDN combines the benefits of both Elite CDN and DACDN. Elite CDN's better elitism and stability coupled with DACDN's DA co-evolution resulted in a configuration that produced the most accurate networks seen thus far, improving upon Elite CDN 10E by a considerable margin.

Elite DACDN does still retain one problem from DACDN. As mentioned earlier, a mutation to a DA individual has the possibility to create a scheme that is no longer label preserving. This is one possibility as to why there exists a number of somewhat steep drops in elite DACDN runs.

6.2.1 Elite DACDN vs DACDN

In figure 7, we compare elite DACDN to DACDN in order to illustrate just how much more effective it really is. As evident in the figure, elite DACDN vastly outperforms DACDN and although DACDN does come close at times, it still falls considerably short in comparison.

DACDN's shortcomings are more evident in figure 7, we can clearly see the inconsistency between its runs, especially in relation to elite DACDN, whose runs are far more stable. However, despite its limitations, DACDN does manage to create networks whose accuracies come close to those in elite DACDN.

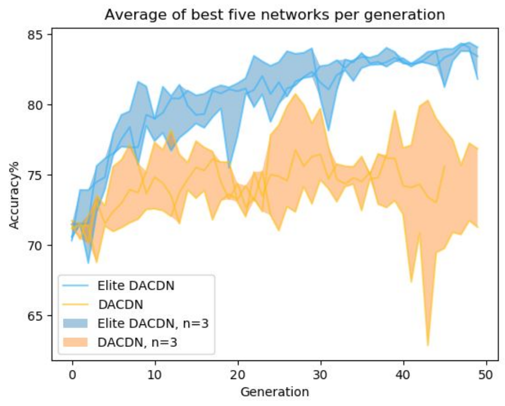


Figure 7: Elite DACDN compared to DACDN

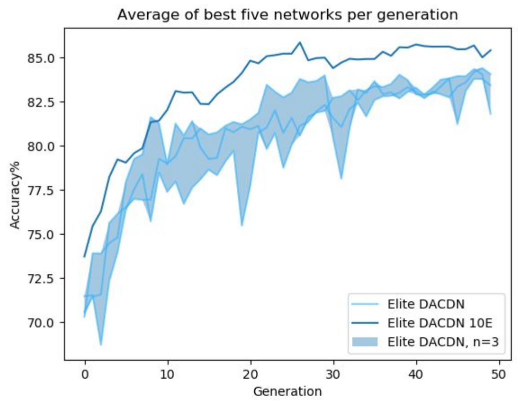


Figure 8: Elite DACDN trained with 5 epochs compared to Elite DACDN trained with 10 epochs

This is proof that despite its flaws, DACDN still has the potential to be a powerful algorithm in its own right.

6.3 Elite DACDN

In an effort to try and and further boost the performance of Elite DACDN. We performed two additional experiments that involved allowing elite DACDN to train its networks for ten and twenty epochs.

Unfortunately, due to a combination of long computation times and time constraints, we could only afford to run each test once. As such, further experimentation to determine the consistency of elite DACDN with more epochs was not possible. However we hypothesise that these results are indicative of general performance.

6.3.1 Ten epoch experiment

In figure 8, we can see the effect of giving Elite DACDN more time to train. Given 10 epochs to train on, elite DACDN gets a significant increase to its performance, managing to consistently evolve networks that are at least two to three percent more accurate than its less trained counterpart.

Something interesting to note is that Elite DACDN makes far better use of additional epochs than Elite CDN or base CDN do.

This is evident in figure 4 and 5, where additional epochs only marginally improved overall performance. Compare that to figure 8 and we can clearly see that DACDN has the ability to better utilise additional training. This is likely because training on the same dataset for longer is worse than training on a dataset that has been augmented for longer. This is corroborated by the fact that DAs are known to help prevent overfitting [3, 4]

6.3.2 Twenty epoch experiment

Given 20 epochs. Elite DACDN managed to improve its performance even further. Unfortunately, due to time constraints, we could only afford to run this test for 22 generations. Despite this, Elite DACDN with 20 epochs managed to achieve the highest evolved accuracy out of all our tests, with its best performing network receiving a 87.69% accuracy. Something important to note is that this run utilised a minor extension referred to as node breeding. The details of this extension can be found in [26].

6.4 Additional Discussion

6.4.1 DA effectiveness

DACDN showcases the utility of using data augmentations in effective ways. DACDN was given the option (no_operation) to produce schemes that simply doubled the data. However, it always
chose to avoid such an outcome. Knowing this, one can infer that the use of DA schemes to inflate data is much more effective than simply training on the same data for longer. This is corroborated by the fact that Both DACDN and Elite DACDN significantly outperformed their non-DA CDN counterparts when trained for the same amount of epochs.

6.4.2 DA Schemes produced

The types of DA schemes that DACDN and elite DACDN converged on is another topic worth addressing. In DACDN, the DA schemes that the best networks from each generation used were radically different from each other. This is likely because DACDN's poor elitism doesn't hold onto performant DA schemes for long. Meaning that the DA schemes evolved are far more random. However the same cannot be said for elite DACDN. In elite DACDN, the flip_lr DA (accompanied by some other DAs) appeared in the DA scheme of almost every top performing network.

Interestingly, the size of the schemes produced by DACDN and elite DACDN were quite different. The top performing networks in DACDN used schemes with three to four DAs whereas Elite DACDN's top performing networks rarely used schemes consisting of more than two DAs. Furthermore, we found that the blueprints that co-evolved with DAs were considerably smaller than those that did not (see appendix C).

However, one must remember that all experiments were performed on the CIFAR-10 dataset. This is important because the top performing DA schemes that were evolved on this dataset might be radically different from those evolved on another.

Configuration	**Best evolved Network**	**Best fully trained network**
Original CDN [1]	<80%	92.3%
Base CDN	75.83%	78.27%
Elite CDN	82.81%	84.89%
DACDN	80.93%	85.08%
Elite DACDN	84.82%	87.08%
Elite DACDN 10E	86.1%	91.11%
Elite DACDN 20E	87.69%	92.12%

Table 9: table depicting the best evolved and fully trained networks for each configuration

6.4.3 Comparison to original CoDeepNEAT

In original CoDeepNEAT [1], the best network they managed to evolve with CIFAR-10 received less than 80% accuracy [1]. This was achieved in 72 generations using 8 epochs. Original CoDeepNEAT also used data augmentations but did not co-evolve them like we have. Additionally, they likely had time to tune their hyper-parameters and squeeze further performance from their algorithm, a luxury which we unfortunately did not have.

Base DACDN evolved its best network with an accuracy of 80.93% which slightly beat out original CDN, this was achieved in 50 generations using 5 epochs. However, once again, it is important to note that DACDN actually trains for double the epochs specified, meaning that it actually used 2 more epochs than original CDN. Additionally, base DACDN's inconsistency in its performance means that with further tuning, future runs could prove to produce better results.

On the other hand, Elite DACDN consistently managed to produce networks that outperformed original CDN during evolution with its top performing evolved network having an accuracy of 84.82%. However, this was also achieved using two more epochs than original CDN. When given 20 epochs to train on, elite CDN managed to improve its performance more and evolve its best network with an accuracy of 87.69%.

6.4.3 Fully Training

In [1], Liang, J et al fully trained their best network after evolution and managed to achieve 92.3% accuracy. We decided to fully train every configuration's best network to see what the maximum accuracy we could achieve was. DACDN and elite DACDN networks were all trained for 150 epochs whereas CDN and elite CDN networks were trained for 300 epochs. Additionally all networks were trained with a feature multiplier of 2. The feature multiplier multiplies the feature count of every layer by a specified value in order to proportionally increase the size of the network, so as to make better use of additional training epochs.

Those networks that evolved with Elite DACDN were trained using the top 3 best DA schemes found during that network's evolutionary run. This proved to illicit better results than just using the best DA scheme. We hypothesise that because the best performing DA schemes evolved by elite DACDN were similar but still different from each other, they aided in eliminating additional dimensions of overfitting. This is corroborated by the fact that base DACDN, whose best DA schemes were quite different from each other, performed badly when trained with its top 3 DA schemes. Furthermore, during limited testing we found that networks that evolved with data augmentations performed consistently worse when trained without their DA scheme on average being 4% to 5% less accurate.

Unfortunately, despite getting close, we were not able to achieve an accuracy higher than original CDN's best fully trained network. The top performing networks both during evolution and after being fully trained can be found in table 9.

6.4.3 Contributions

DACDN and Elite DACDN are the first instances of a co-evolutionary approach to automated data augmentation. In this paper we have proven that such an approach has the potential to be extremely useful in increasing the performance of evolved networks. Keep in mind that this is without any tuning or further experimentation with the DA population. However, despite our success, results indicate that there is still plenty of room to further explore co-evolving DA schemes within the field of neuro-evolution.

DACDN and elite DACDN have the capability to track which DAs performed the best within an evolutionary run. For instance with elite DACDN we tracked that flip_lr was very common within the DA schemes of the best networks. Information such as this, could be valuable to determine which DA schemes or even individual DAs work well with specific types of networks and datasets.

Additionally, during our training of the best performing evolved networks that elite DACDN produced we used the top 3 best DA schemes that they found to increase performance. We found these DA schemes all increased the accuracy of these networks during training. From this we can infer that elite DACDN has the capability to evolve DA schemes that not only perform well with the blueprint they evolved with but also with other DNNs within the population.

7 FUTURE WORK

There are multiple different improvements and tests that can still be done in the future. There is plenty to be gained from tuning the hyperparameters of CoDeepNEAT in the effort to find a better configuration than we have used.

The method whereby we train on augmented data, specifically the epoch by epoch system, may be a rather naive approach. As such, there is room to explore alternatives.
Something simple worth experimenting with is the DA population itself. There exists countless different DAs created over the years and although we have found success using the ones we picked (see appendix B) it is entirely possible to select a population of DAs that result in different but potentially more effective DA schemes.

An extension worth implementing is the ability to create non-deterministic DA schemes. By non-deterministic we mean that a scheme has a probability to apply a certain DA in the scheme or not. Instead of using a linear directed graph to represent DA schemes, a branching directed graph could be used with each connection representing a probability.

8 CONCLUSION

DACDN, despite taking longer to compute (approximately 2.5 times longer) has shown to considerably increase the performance of base CDN. Its extension, elite DACDN has proven to not only illicit even better results but also make far better use of editional epochs. This makes it a more than worthy extension to include in any implementation of CDN. Although we failed to achieve state-of-the-art results, with further experimentation and tuning, results indicate that DACDN and elite DACDN likely have the potential to achieve very impressive results in the future. DACDN and Elite DACDN show that our novel approach to automated augmentation, namely the coevolution of DA schemes, is a promising solution to incorporating DA into AutoML.

9 ACKNOWLEDGEMENTS

We would like to thank our supervisor Geoff Nitschke for all the guidance and support given throughout the making of this report. Thanks to Shane Acton and Sasha Abramowitz for their valuable help in implementing CoDeepNEAT and all its extensions. We would also like to give credit to GitHub user aleju for his data augmentation library which was used extensively throughout the project.

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] Taylor, L. and Nitschke, G., 2018, November. Improving Deep Learning with Generic Data Augmentation. In *2018 IEEE Symposium Series on Computational Intelligence (SSCI)* (pp. 1542-1547). IEEE.

[4] Wang, J. and Perez, L., 2017. The effectiveness of data augmentation in image classification using deep learning. *Convolutional Neural Networks Vis. Recognit*.

[5] Stanley, K.O., Clune, J., Lehman, J. and Miikkulainen, R., 2019. Designing neural networks through neuroevolution. Nature Machine Intelligence, 1(1), pp.24-35.

[6] 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).

[7] Zurada, J.M., 1992. Introduction to artificial neural systems (Vol. 8). St. Paul: West publishing company.

[8] LeCun, Y., Bengio, Y. and Hinton, G., 2015. Deep learning. nature, 521(7553), p.436.

[9] Forsyth, D.A. and Ponce, J., 2002. Computer vision: a modern approach. Prentice Hall Professional Technical Reference.

[10] 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).

[11] Back, T., 1996. Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms. Oxford university press.

[12] Hecht-Nielsen, R., 1992. Theory of the backpropagation neural network. In Neural networks for perception (pp. 65-93). Academic Press.

[13] Hawkins, D.M., 2004. The problem of overfitting. Journal of chemical information and computer sciences, 44(1), pp.1-12.

[14] Wang, J. and Perez, L., 2017. The effectiveness of data augmentation in image classification using deep learning. Convolutional Neural Networks Vis. Recognit.

[15] Lemley, J., Bazrafkan, S. and Corcoran, P., 2017. Smart augmentation learning an optimal data augmentation strategy. IEEE Access, 5, pp.5858-5869.

[16] Cubuk, E.D., Zoph, B., Mane, D., Vasudevan, V. and Le, Q.V., 2018. Autoaugment: Learning augmentation policies from data. arXiv preprint arXiv:1805.09501.

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

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

[19] Zhang, H., Kiranyaz, S. and Gabbouj, M., 2018. Finding Better Topologies for Deep Convolutional Neural Networks by Evolution. arXiv preprint arXiv:1809.03242.

[20] Canuma, P. (2019). What problems can we solve using AI?. [online] Towards Data Science. Available at: https://towardsdatascience.com/what-problems-can-we-solve-using-ai-ec7131f8159b [Accessed 1 May 2019].

[21] Young, S. (2019). 10 trends of Artificial Intelligence (AI) in 2019. [online] Becoming Human: Artificial Intelligence Magazine. Available at: https://becominghuman.ai/10-trends-of-artificial-intelligence-ai-in-2019-65d8a373b6e6 [Accessed 1 May 2019].

[22] Kuznichov, D., Zvirin, A., Honen, Y. and Kimmel, R., 2019. Data Augmentation for Leaf Segmentation and Counting Tasks in Rosette Plants. arXiv preprint arXiv:1903.08583.

[23] Hussain, Z., Gimenez, F., Yi, D. and Rubin, D., 2017. Differential data augmentation techniques for medical imaging classification tasks. In AMIA Annual Symposium Proceedings (Vol. 2017, p. 979). American Medical Informatics Association.

[24] 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).

[25] Verdee, A. (2018). Data Augmentation Experimentation. [online] Medium. Available at: https://towardsdatascience.com/data-augmentation-experimentation-3e274504f04b [Accessed 30 Aug. 2019].

[26] Acton, S. 2019. ‘Improving CoDeepNEAT with elitism and stability’. Honours. University of Cape Town. Cape Town

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

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

[29] 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.

[30] 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.

APPENDIX

A System Information

Implementation details

The pytorch framework[4] as used for the creation of all neural networks and was largely chosen for both its ease of use and its GPU compatibility.

Parallelism

Due to the nature of a system such as this one, it was imperative to utilise parallelism in order to speed up the lengthy computation times. To illustrate how long computation times can be, DACDN, when training on 5 epochs, can take approximately 2 hours to complete a single generation with one GPU. As such, the system was developed with the aim of utilising multiple GPUs for experimentation.

We managed to achieve massive speedup by training each assembled network on its own individual GPU, which should be an embarrassingly parallel process. When using multiple GPUs, we observed that DACDN now took slightly less than one hour per generation when using 4 GPUs.

Mutation chances for CDN evolvable parametres

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%
Percent to reproduce	20%
Elite to keep	10%
Evaluations per generation	75
Layer size mutation chance	22%
Layer type mutation chance	8%

[4] https://pytorch.org/

B Data Augmentation

Data augmentations library

The open source library imgaug[5] was used for all data augmentations used in DACDN (except for Canny_Edges)

Data Augmentation Population

Augmentation	**Description**
Flip_lr	Flips image horizontally
Rotate	Rotates image
Translate_Pixels	Translate image along the x-axis or y-axis
Scale	Scale image to percentage of its original size in either its x or y dimensions
Pad_Pixels	Pads image i.e adds rows or columns to it by a number of pixels
Crop_Pixels	Crops image i.e removes rows or columns from it by a number of pixels
Canny_Edges	Extracts Canny Edges from image
Additive_Gaussian_Nois e	Adds gaussian noise (aka white noise) to an image
Course_Dropout	Randomly erases portions of the image
HSV (Colour Jittering)	Increase each pixel's Hue (H), Saturation (S) or Value/lightness (V)
Grayscale	Converts image to grayscale
No_Operation	Performs no augmentation (image is unchanged)

[5] https://github.com/aleju/imgaug

Evolvable Parameters in DA Population

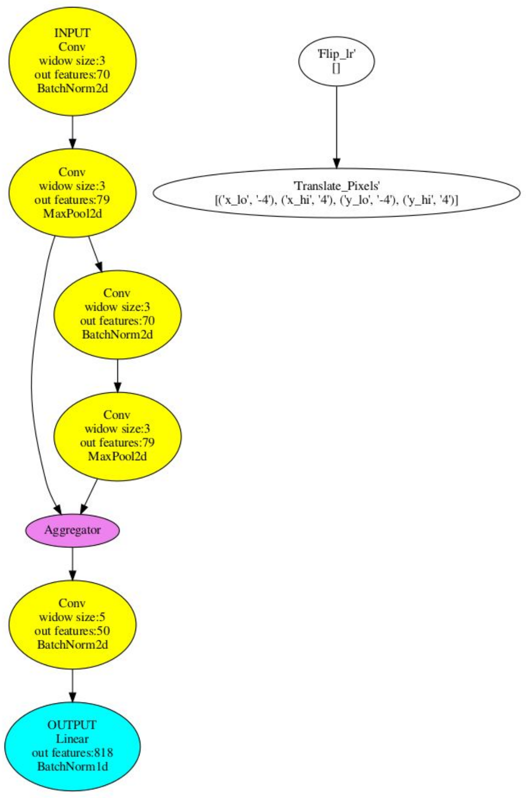
Augmentation	**Evolvable parameters**	**Initialized values**
Flip_lr	**None**	**None**
Rotate	**lo:** lower bound on degree range **hi:** upper bound on degree range	**lo:** -30 **hi:** 30
Translate_Pixels	**x_lo:** lower bound on x-axis range **y_lo:** upper bound on x-axis range **y_lo:** lower bound on y-axis range **y_lo:** upper bound on y-axis range	**x_lo:** -4 **x_hi:** 4 **y_lo:** -4 **y_hi:** 4
Scale	**x_lo:** lower bound on x-axis range **y_lo:** upper bound on x-axis range **y_lo:** lower bound on y-axis range **y_lo:** upper bound on y-axis range	**x_lo:** 0.75 **x_hi:** 1.25 **y_lo:** 0.75 **y_hi:** 1.25
Pad_Pixels	**lo:** lower bound on pad range **hi:** upper bound on pad range **s_i:** decides whether a value is sampled independently for each edge or remains the same	**lo:** 1 **hi:** 4 **s_i:** False
Crop_Pixels	**lo:** lower bound on crop range **hi:** upper bound on crop range **s_i:** decides whether a value is sampled independently for each edge or remains the same	**lo:** 1 **hi:** 4 **s_i:** False
Canny_Edges	**min_val:** edges with intensity gradient smaller than min_val are definitely not edges (non-edge) **max_val:** edges with intensity gradient larger than max_val are definitely edges (sure-edge) **Note**: Edges in between min_val & max_val are considered edges if they are connected to "sure-edge" pixels	**min_val:** 100 **max_val:** 200
Additive_Gaussian_Noise	**lo:** lower bound on value range **hi:** upper bound on value range **percent:** value is sampled once for percent of images and sample three (RGB) times (channel wise) for the rest	**lo:** 0.05 **hi:** 0.125 **Percent:** 0.6
Course_Dropout	**d_lo:** lower bound on percent of pixels to drop **d_hi:** upper bound on percent of pixels to drop **s_lo:** lower bound on scaling factor **s_lo:** upper bound on scaling factor **percent:** in percent of images only the information of some channels is set to 0 while others remain untouched	**d_lo:** 0.05 **d_hi:** 0.2 **s_lo:** 0.025 **s_hi:** 0.5 **percent:** 0.6
HSV (colour jittering)	**channel:** determines which HSV channel is being manipulated (0:Hue, 1: Saturation, 2: Value) **lo:** lower bound on value range **hi:** upper bound on value range	**channel:** 0 **lo:** 20 **hi:** 50
Grayscale	**a_lo:** lower bound on alpha value **a_hi:** upper bound on alpha value	**a_lo:** 0.35 **a_hi:** 0.75
No_Operation	**None**	**None**

Mutation chances for DA evolvable parametres

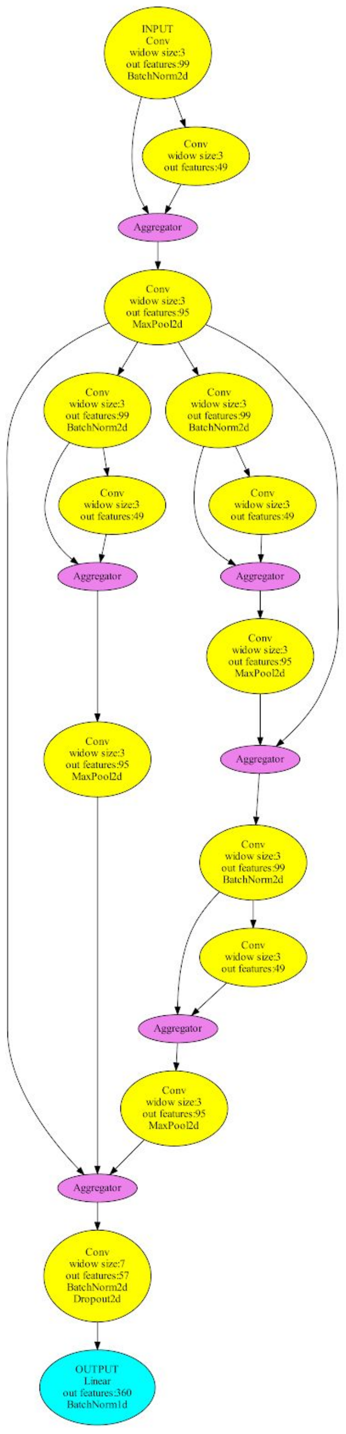
Value Type	**Mutation Chance**
Whole Values	20%
Continuous Values	30%
Discrete Values	10%
Boolean Values	20%
DA ignore chance	20%

C Best performing networks

The best performing Elite DACDN network alongside its co-evolved DA


INPUT Conv widow size:3 out features:70 BatchNorm2d
Conv widow size:3 out features:79 MaxPool2d
Conv widow size:3 out features:70 BatchNorm2d
Conv widow size:3 out features:79 MaxPool2d
Aggregator
Conv widow size:5 out features:50 BatchNorm2d
OUTPUT Linear out features:818 BatchNorm1d
'Flip_lr' []
'Translate_Pixels' [('x_lo', '-4'), ('x_hi', '4'), ('y_lo', '-4'), ('y_hi', '4')]

The best performing Elite CDN network


INPUT Conv widow size:3 out features:99 BatchNorm2d
Conv widow size:3 out features:49
Aggregator
Conv widow size:3 out features:95 MaxPool2d
Conv widow size:3 out features:99 BatchNorm2d
Conv widow size:3 out features:99 BatchNorm2d
Conv widow size:3 out features:49
Conv widow size:3 out features:49
Aggregator
Aggregator
Conv widow size:3 out features:95 MaxPool2d
Conv widow size:3 out features:95 MaxPool2d
Aggregator
Conv widow size:3 out features:99 BatchNorm2d
Conv widow size:3 out features:49
Aggregator
Conv widow size:3 out features:95 MaxPool2d
Aggregator
Conv widow size:7 out features:57 BatchNorm2d Dropout2d
OUTPUT Linear out features:360 BatchNorm1d