Comparing Evolutionary Algorithms for Deep Neural Networks

Alan Zhao, Harsha Uppili and Gabriel Meyer-Lee
Swarthmore College, Swarthmore, PA 19081
e-mail: {huppili1, zzhao1, gmeyerl1}@swarthmore.edu

Abstract

Genetic algorithms have long been successfully applied to optimize the weights of neural networks. The genetic concept of crossover, recombining components to form a stronger component, applies well to Deep Neural Networks. This paper elaborates on the performances of two automated evolutionary methods for optimizing deep learning architectures on the relevant and important tasks of image classification and language modeling. The first of these methods is based on evolving populations of chromosomes and crossing over the strongest amongst them, and the second is based on a hierarchical genetic representation scheme. Both of the algorithms required a substantial amount of computational power in the original literatures. Given the limited resources available, presented are the implementations of these algorithms at a relatively small scale for both tasks as well as results on the CIFAR-10 dataset, demonstrating classification quality comparable to that of literature and setting the stage for the future work of merging these methodologies.

1 Introduction

With the advent of big data and superior hardware, scaling very powerful machine learning systems has become more realistic than ever before. Thus, there has recently been an influx of advances in training Deep Neural Networks (DNN), specifically Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN), to tackle a multitude of problems in computer vision, language processing, and many other domains.

A challenge of widespread adaption of DNNs is that the most successful networks are the ones that often are the most complex, with extremely intricate topologies to sort through and hundreds of hyperparameters to optimize. Given the difficulty of the task, alternative methods of identifying network architectures and optimizing the hyperparameters have been sought after. Such methods are thought of as preferable over relying on human engineers to make the precise design choices necessary to build the strongest networks. One of these alternative methods would be for the human engineer to do the high-level organizational work, and for the computer to do the bulk of the work on identifying an optimal network architecture.

This paper demonstrates the success of two evolutionary algorithms on choosing optimal network architectures for evolving CNNs for the task of image classification and LSTMs for the task of language modeling. We demonstrate algorithms that utilize hierarchical genetic representation schemes and incremental increased topology complexities to evolve networks comparable to existing literatures.

2 Background

The field of neuroevolution, which refers to applying Evolutionary Algorithms to evolving neural networks, emerged in the 90's as a way of creating neural networks prior to the rise of GPU-driven supervised learning. Methods were developed such as SANE [11], which evolved the synaptic weights of the network, while another class of methods was developed to evolve the structures of the neural networks. While the former has largely been overshadowed by the achievements of backpropagation and gradient descent, the latter lives on thanks to the development of Neuroevolution of Augmenting Topologies (NEAT) [14]. As further discoveries like ReLU activation functions and batch normalization, as well as advances in GPU design allowed the construction of deeper and larger neural networks, a new field of "meta-learning" emerged. Meta-learning utilizes learning techniques to optimize the hyperparameters of deep neural

networks (DNNs), generally either using convolutional layers for image classification or LSTMs for language modeling. Recent work has shown impressive results in the field of meta-learning on benchmark datasets with Bayesian Optimization [13], Reinforcement Learning [15], and Evolutionary Algorithms [12].

CoDeepNEAT [10] is an algorithm developed recently by Mikkulainen et. al. which marries the fields of neuroevolution and meta-learning. CoDeepNEAT is an extension of NEAT to DNNs. Each node gene in the chromosome is no longer a single neuron and is instead treated as an entire layer in a DNN with hyperparameters determining both the type of layer (recurrent, dense, etc.) and that layer's properties. The chromosome also tracks the connections between layers, although these connections are no longer evolved as "genes" and their weights are determined through supervised learning. Each chromosome is evaluated by being compiled into a network, trained for a fixed number of epochs, and assigned a fitness equal to it's validation accuracy.

The macro-view of the CoDeepNEAT algorithm involves two main components: modules and blueprints. Blueprint chromosomes are made up of genes representing a linearly connected set of modules. Module chromosomes are made up of genes representing an interconnected set of DNN layers. The key innovation of CoDeepNEAT is that these two interconnected components are represented as separate populations which are evolved in parallel. The fitnesses of the modules are determined stochastically from the fitnesses of the blueprints of which they are a part. One key aspect of CoDeepNEAT that was largely unaddressed within the paper is the structure of the software implementations of the blueprints and modules. The authors, in fact, did not settle on a single implementation; instead, they came up with distinct implementations for each different benchmark test that CoDeepNEAT was demonstrated on, essentially declining to address the issue of how exactly a DNN should be encoded for evolution.

Thankfully, recent research [3] tackles this exact problem. One particular study focuses on developing hierarchical representations of DNNs, specifically representations of deep convolutional neural networks. The core concept of this representation is that at every potential hierarchical level, a DNN can be represented as a directed acyclic graph where each node is a tensor (or "feature map") and each edge represents some nonlinear operation. At the lowest level, these operations are considered to be neural network layers, in this case drawn from a fixed list of convolutional layers considered to be "primitive operations." The key insight behind this fixed list is that a wider variation of layers can be created from an arrangement of simple fixed layers. For example, two parallel convolutional layers with 16 filters each is exactly equivalent to a single convolutional layer with 32 filters. The authors of this paper demonstrate the usefulness of this representation on image classification benchmarks using both a random search and a simple evolutionary scheme.

3 Implementation

3.1 CoDeepNEAT

Our implementation of CoDeepNEAT is based largely on the given description of the application of CoDeepNEAT to image classification with the CIFAR-10 dataset [1] and to language modeling with the Penn Tree Bank (PTB) dataset [9]. We wrote our implementation in Python3.5 on top of both the open source NEAT3 code [2] and the Keras deep learning API with a TensorFlow backend. Keras is a widely used and user-friendly API that supports a range of backend tensor operation libraries, thus allowing our implementation to be highly portable. The code describing the Populations and Species from NEAT3 was largely preserved in our implementation, as these components function nearly identically in CoDeepNEAT as they do in NEAT. Two major changes were introduced: an entirely new encoding scheme for the chromosomes and genes, and the evolution routine itself was modified to support coevolution between the blueprints and the modules. The relationship between the encodings of blueprint and module chromosomes is shown below in Figure 1.

Each blueprint chromosome contains both an ordered list of module genes and a dictionary of global hyperparameters. Each of the module genes is a pointer to a species of module chromosomes, which must be checked after each time the module population is speciated to avoid pointing to a nonexistent species. When a blueprint is compiled into a network, a representative module for each species referred to in the blueprint's genome is sampled from a uniform distribution over the species. Thus, two module genes pointing to the same module species will be replaced with the

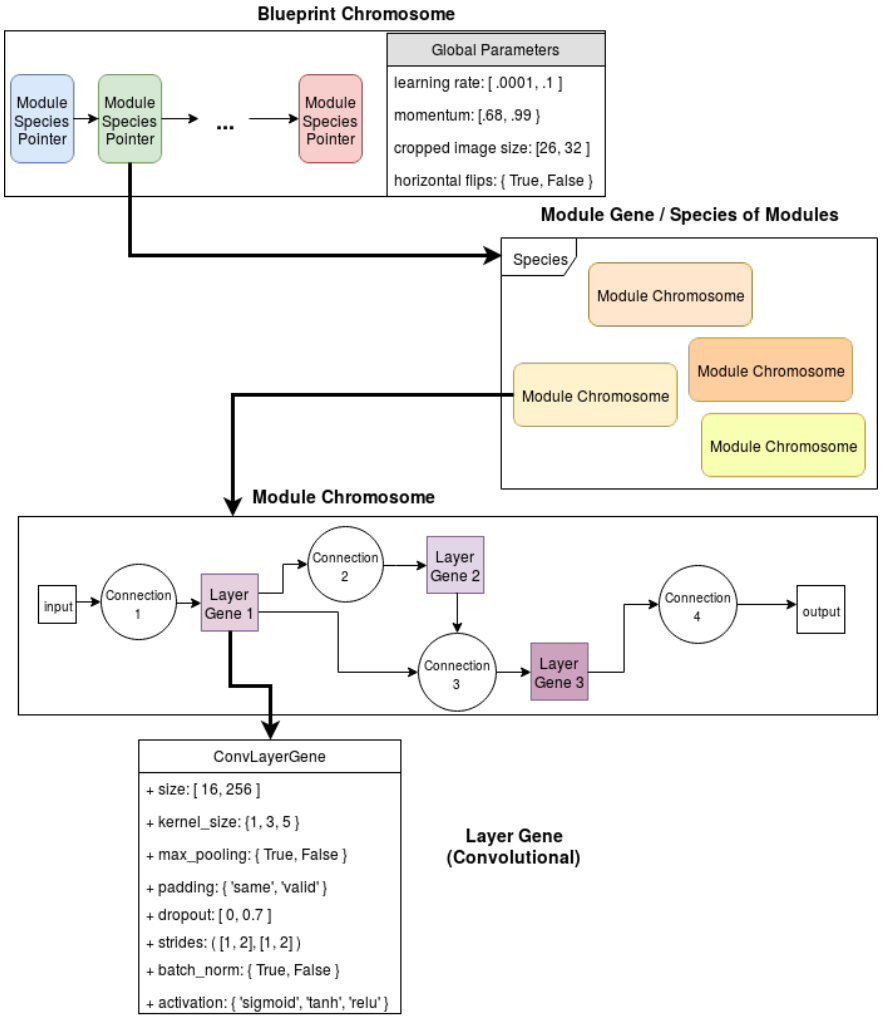


Figure 1: Design of CoDeepNEAT chromosomes and genes

same module every time a network is generated, but the identity of that module can vary for every new network. The blueprint chromosome includes two forms of mutations: inserting a new module gene at a random index and randomly reassigning the pointer of a module gene.

Each module chromosome tracks a set of layer genes and the connections between each of those genes. Our module chromosome supports two forms of mutation: adding a layer gene between two randomly chosen points and mutating individual layer genes. The connections are shown as nodes in the graph given in Figure 1 but in practice encode both the edges of that graph and the merging of tensors that must be performed for any layer receiving more than one input. The merge method concatenates along the last dimension, which in turn is equal to the number of channels for convolutional layers' outputs and the number of outputs for recurrent and dense layers. Convolutional layers' outputs are downsampled via max-pooling to the smallest of the spatial dimensions in the merge. The layer genes encode a either a Conv2D, LSTM, or Dense Keras layer, storing the the parameters for that layer in a dictionary. Figure 1 shows the available range of parameter settings, but each individual layer gene's dictionary actually stores the value used for that layer. These parameters can all be modified through mutation.

```
CoDeepNEAT(maxGenerations, numNetworks, data)
INITIALIZE(modulePopulation)          // as randomly parametrized simple modules, speciate
INITIALIZE(blueprintPopulation)       // as random small blueprints, speciate
generation = 0
while generation < maxGenerations
    for i = 0 to numNetworks
        bp = RANDOM SAMPLE(blueprintPopulation)
        model = ASSEMBLE(bp)
        accuracy = FIT(model, data)
        bp.fitness = bp.fitness + accuracy
        bp.numUse = bp.numUse + 1
        for module in bp
            module.fitness = module.fitness + accuracy
            module.numUse = module.numUse + 1
    for indiv in blueprintPopulation + modulePopulation
        module.fitness = module.fitness ÷ module.numUse
    mparents = TOURNAMENT SELECT(allModuleSpecies)
    mchildren = REPRODUCE(mparents)          // includes crossover and mutation
    modulePopulation = mchildren
    SPECIATE(modulePopulation)
    bparents = TOURNAMENT SELECT(allBlueprintSpecies)
    mbchildren = REPRODUCE(bparents)         // includes crossover and mutation
    blueprintPopulation = bchildren
    SPECIATE(blueprintPopulation)
    generation = generation + 1
```

The above pseudocode shows the changes made to NEAT to support coevolution. Recall that both two populations are tracked instead of one and that the populations are involved in parallel, analogous to the evolution in NEAT, with the exception of the module population, which must first be updated to avoid invalid module species pointers, and the fitnesses are assigned to the each individual probabilistically from the fitnesses of a number of generated networks. Our estimator for the fitness metric of each blueprint and module sets the fitness as the average fitness of the networks that a particular blueprint or module was used in. Unused individuals are given a fitness .01 higher than the average population fitness in order to encourage using them in the next generation.

3.2 Hierarchical Representation

Our implementation of the hierarchical representation of neural networks was also written in Python3.5 on top of Keras. The representation consists entirely of Motifs, which encode a directed acyclic graph. Each Motif maintains an adjacency matrix containing the index of each of the operations represented as the edges between the nodes in the graph (which represent tensors). Operations for a level 2 Motif are pre-defined primitives. The primitives used are given in Table 1. Separable convolution is depthwise-separable convolution as defined in [5]. The recurrent layer is a fully connected layer which receives its outputs at the previous time step as additional inputs while GRU and LSTM both refer to specific recurrent designs with gated access to a hidden memory cell. The Keras implementations are based on the original GRU definition [4] and the original LSTM definition [7]. Operations at higher levels are a set of Motifs at the previous level. We defined two types of architectures, a flat architecture which contains a single level 2 Motif (in our experiments this had 11 nodes) and a hierarchical architecture which contains a single top level Motif and a set number of lower level motifs (in our experiments this was a level 3 Motif with 5 nodes and 6 level 2 Motifs with 4 nodes each).

The above Figure 2 shows the assembly of a level 2 Motif (bottom) and level 3 Motif (top). Merging is assigned as last-dimension concatenation, which does not require down-sampling as all operations within Motifs are padded as to preserve spatial dimensions in convolutional networks. In order to be assembled and trained, the architectures are used as cells in the generic architectures shown in Figure 3 for image classification and a simple single-cell model for language modeling. The single model includes a Keras Embedding layer to embed the input into appropriately sized

Convolutional Networks	Recurrent Networks
Convolution with 1x1 kernel and C filters	Recurrent layer with C output nodes
Convolution with 3x3 kernel and C filters	GRU layer with C output nodes
Separable Convolution with 3x3 kernel and C filters	LSTM layer with C output nodes
Max Pooling with 3x3 kernel and 1x1 stride	Max Pooling with 3 kernel and 1 stride
Average Pooling with 3x3 kernel and 1x1 stride	Average Pooling with 3 kernel and 1 stride
Identity	Convolution with 3 kernel and C filters
No Op	Identity
	No Op

Table 1: Hierarchical primitive operations

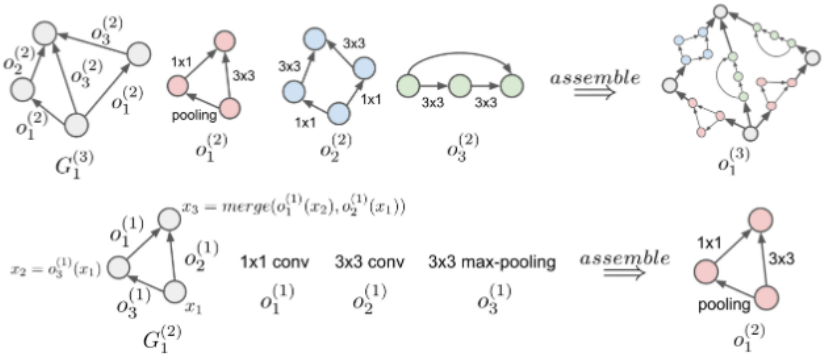


Figure 2: Hierarchical representation of convolutional network [3]

vectors, then the cell, then a Dense layer with softmax activation to produced scaled output vectors.

Mutation on an architecture involves sampling a random level (> 1), then sampling a random Motif at that level, then sampling a random successor node in the Motif, then sampling a random predecessor node in the Motif, all of which are over uniform probability. The operation between these two nodes is reassigned randomly from the list of available operations over a uniform probability distribution. Architectures were initialized by setting each Motif in the architecture to a chain of identity operations, then applying a large number of mutations to the architecture. The Evolutionary Algorithm used to evolve these architectures was much simpler than CoDeepNEAT, and used tournament selection with $k = .05 \times popsize$ instead of $k = 2$ which returns a copy of the most fit Motif in the tournament. Pseudocode for the hierarchical evolution is given below. This algorithm performs evolution in steps, where each step generates a single child, rather than in generations and does not include speciation.

```
HIERARCHYEVOLVE(numSteps, population, data)
INITIALIZE(population)                    // large number of random mutations
step = 0
while step < numSteps
    while step < population.size
        arch = population[step]
        model = ASSEMBLEASSMALLMODEL(arch)
        accuracy = FIT(model, data)
        arch.fitness = accuracy
    parent = TOURNAMENT SELECT(evaluatedPopulation)
    child = MUTATE(parent)                // crossover is not used for this evolution
    model = ASSEMBLEASSMALLMODEL(child)
    accuracy = FIT(model, data)
    child.fitness = accuracy
    population.APPEND(child)              // population increases with each step
    step = step + 1
```

4 Experiments

In order to examine the performance and adaptiveness of our implementation, we performed experiments on both image classification and language modeling benchmark datasets. In the following subsections, the benchmark datasets are briefly introduced and the experimental setup for both CoDeepNEAT and Hierarchical Representation implementations are documented.

4.1 Image Classification

In our image classification experiments, we used the CIFAR-10 dataset. The CIFAR-10 dataset contains colored 32*32 images (3 channels) in 10 different categories. It has 50,000 images in the training set and 10,000 images in the testing set. To compare our results with, we also built a ResNet-18 [6] implementation and tested its performance under the same settings as the benchmark.

4.1.1 CoDeepNEAT

For our CoDeepNEAT experiment, we evolved the topology of a CNN to maximize its classification performance on the CIFAR-10 dataset. We initialized the blueprint population with size of 15 and the module population with size of 10. From those two populations, 25 CNNs were assembled for fitness evaluation in every generation. During fitness evaluation, the training set was divided into a 42,500 image training subset and 7,500 image validation subset so that the testing dataset was unseen to the evolution process. Given that training deep neural networks could be computationally expensive and our limited resources, we forwent the global parameters evolution and turned off mutation for some of the layer parameters to prioritize evolution of the architecture. Each model was trained for 8 epochs using the default Adam optimizer settings. The data augmentation included horizontal and vertical shifts with a factor of 0.1 and random horizontal flips of the images. After training, the validation subset was used to determine the fitness of the models. After the evolution, the best performing models of each generation was trained on the entire training set (50,000 images) and evaluated on the test set (10,000 images). Table 2 shows that the parameter settings used in the experiments. Table 3 shows the layer parameters that could be mutated during mutation.

4.1.2 Hierarchical Search

For our Hierarchical Search experiment, both flat and hierarchical representation of architectures were used for random search and hierarchical representation was used for random search, evolutionary search and random generation.

For the random search, we randomly generated 50 modules using the hierarchical representation. Then we performed 100 mutations on each generated modules to obtain the initial population. We used a small model showed in Fig. 3a for fitness computation for efficiency. Each module in the initial population was inserted into the small model and

Parameter	Setting
Generations	25
Module Population	10
Blueprint Population	15
Model Population	25
Input nodes	32, 32, 3
Output nodes	10
Prob. to mutate layer	0.3
Prob. to add a layer	0.1
Prob. to add a Conv. layer	1.0
Prob. to add module in a blueprint	0.05
Prob. to switch to a different module	0.1
Elitism	1

Table 2: CoDeepNEAT parameter settings used in the experiments.

Conv. Parameter	Range
Number of Filters	[16, 256]
Max Pooling	{True, False}
Kernel Size	{True, False}
Dropout Rate	[0, 0.7]
Dense Parameter	**Range**
Output nodes	[16, 256]
Dropout Rate	[0, 0.7]

Table 3: layer parameters available for mutation.

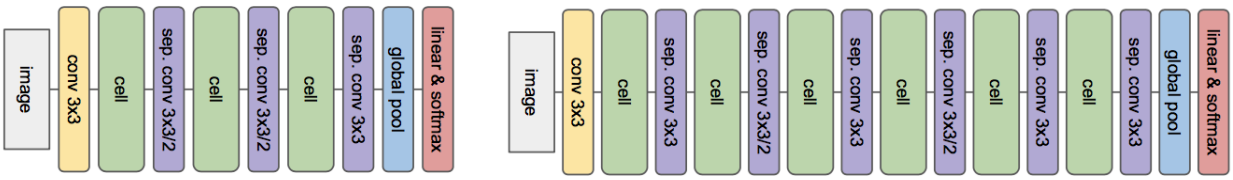
trained on a training subset of 40,000 images. For data augmentation, each training image was randomly cropped to 24*24 and randomly horizontally flipped. The fitness of each module was evaluated on the 10,000 image validation subset. The module with the highest fitness among the 50 was inserted into a large model showed in Fig. 3b. The assembled model was then trained with the entire training set for prolonged epochs until plateau and evaluated on the test set.

For the evolutionary search, we followed the same procedure as random search. In addition, after evaluating the fitnesses of the initial population, 5% of the population were randomly picked and the module with the highest fitness in the 5% went through a mutation. The mutated module was evaluated on the validation subset and added to the population. We set the evolution to have 150 steps so the final population size was 200. After the evolution, the module with the highest fitness overall was inserted into the large model showed in Fig. 3b. The model was trained with the entire training set and evaluated on the test set.

For the random generation, we randomly generated an architecture using the hierarchical representation, and performed 100 mutations on the initial architecture. Then the mutated architecture was inserted into the large model. The model was trained with the entire training set and evaluated on the test set.

4.2 Language Modeling

Language modeling is a fundamental task in artificial intelligence. A language model simply is a probability distribution over a sequence of words or characters and is used to predict the next word in the sequence. We modeled the Penn Tree Bank (PTB) dataset, a widely used corpus with 10000 training words in its vocabulary. The training set contains 929,000 words, the validation set 73,000 words, and the test set contains 82,000 words [8].



(a) small model

(b) large model

Figure 3: Two models used for CIFAR-10 [3]

4.2.1 Evaluating Language Models

Let us denote a given sequence of n words in the training corpus to be

$$w_1 : n$$

Training the LSTM involves minimizing the negative log-likelihood of the sequence

$$NLL = -\sum_{n=1}^{N} logP(w_n|w_{1:n-1})$$

The Perplexity of the model of the sequence, given below, is calculated over the test set. [8]

$$PPL = exp(NLL/T)$$

4.2.2 CoDeepNEAT

For the sole CoDeepNEAT run on the PTB, We evolved the topology of a CNN to maximize its perplexity on the PTB test set. We initialized the blueprint population with size of 15 and the module population with size of 10. The PTB validation set was used for fitness evaluation, allowing for saving the test set. Given that training deep neural networks could be computationally expensive and our limited resources, each model was trained for 3 epochs using the default Adam optimizer. After the evolution, the best performing models of each generation was trained on the entire training set and evaluated on the PTB test set.

4.2.3 Hierarchical Search

For the sole evolutionary search conducted on the language modeling task, we randomly generated 10 modules using the hierarchical representation. Then we performed 20 mutations on each generated modules to obtain the initial population. LSTM variants of the small model seen in Figure 1a are representative of the generated modules. Analogous to the process for Image Classification, each module in the initial population was inserted into the small model and trained on PTB training set. The fitness of each module was then evaluated on the validation set. After evaluating the fitnesses of the initial population, 5% of the population would be randomly picked and the module with the highest fitness in the 5% would go through a mutation. The mutated module would be evaluated and added to the population. After 50 steps of evolution, the module with the highest fitness overall was inserted into a large model, which in turn was trained with the entire training set and will be evaluated on the test set.

5 Results

5.1 Image Classification

We introduce the results for both CoDeepNEAT and Hierarchical Search, and compare the two algorithms in terms of performance and computational complexity.

LSTM Parameters	Fixed
Max Sequence Length	35
Batch Size	20
Output nodes	49
Loss	Sparse Categorical Entropy

Table 4: Fixed parameters for Language Modeling for both Hierarchical Search and CoDeepNEAT

5.1.1 CoDeepNEAT

The best network for CoDeepNEAT was found in the 24th generations, as shown in Figure 4. The network reached 86.33% of test accuracy after 50 epochs of training. In Figure 5, the red line shows the best fitness and the blue line shows the average fitness. As shown in Figure 4 and Figure 5, the evolution did show improvements over generations. This pattern was more observable for the average fitnesses across generations (blue line). The blueprint fitness and module fitness were largely similar to the best test accuracy due to the aforementioned fitness assignment scheme.

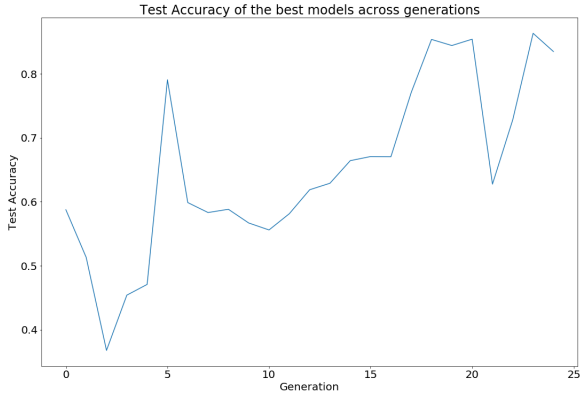


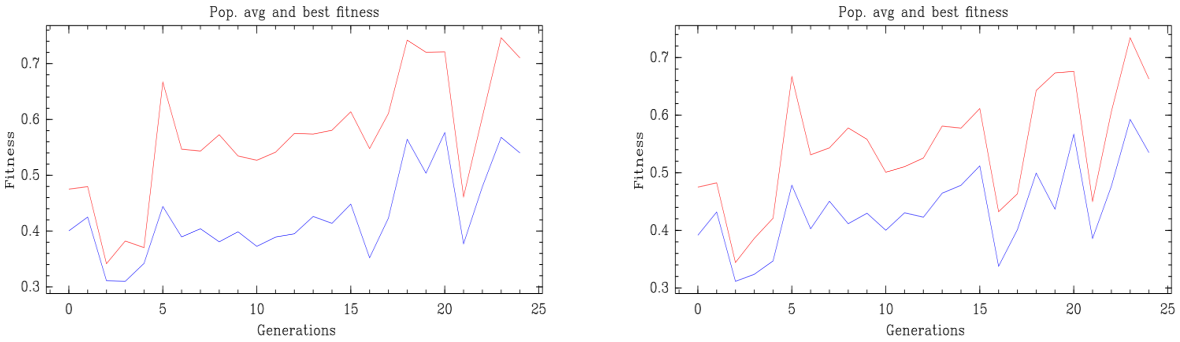
Figure 4: The test accuracies of the best models across generations

5.1.2 Hierarchical Search

We noticed that the flat representation produced larger networks than the hierarchical representation did, as shown in Table 5 and we did not have the resources to fully evaluate the best performing network from the flat representation experiment. The results showed that the flat representation was better at generating better-performing yet highly complex architectures. More interestingly, as shown in Table 6, both CoDeepNEAT and Hierarchical Search were able to evolve architectures with competitive performances compared to the benchmark. Hierarchical Search, in particular, was able to find light architectures with high performance.

Experiment	Avg. Validation Accuracy	Parameters(M)
Flat Representation, random search (50 samples)	0.7347 ± 0.0464	1.059 ± 0.550
Hier. Representation, random search (50 samples)	0.6303 ± 0.0940	0.246 ± 0.137
Hier. Representation, evolution steps (200 steps)	0.6508 ± 0.0691	0.337 ± 0.160

Table 5: Average results from each experiment



(a) Blueprint fitness

(b) Module fitness

Figure 5: Average and highest fitness of Blueprint and Module populations across generations

Best model	**Test Accuracy**	**Parameters(M)**
CoDeepNEAT	0.8633	18.128
Hier. Representation, random sampling	0.8535	0.666
Hier. Representation, random search (50 samples)	0.8927	0.652
Hier. Representation, evolution steps (200 steps)	0.8943	1.174
ResNet-18 [6]	86.75	11.192

Table 6: Best performing models from each experiment

5.2 Language Modeling

Despite our confidence in the evolutionary algorithms presented in this work, the ability of those algorithms to produce literature-comparable language modeling results necessitates both profound computational power and time, both of which we did not have at our disposal at a large enough degree to complete the sole run of each algorithm on the PTB. Thus, no conclusive results for the language modeling task are presented at this time.

6 Discussion

The results of this paper show that both multi-layered coevolutionary approaches and hierarchical approaches to optimizing DNNs are feasible and can develop DNNs comparable to those with hand-designed architectures. We demonstrate results comparable to benchmark models on the CIFAR-10 image classification dataset and present a ready-to-go platform for evolving LSTMs to tackle language modeling. The latter in particular, was not achieved by either of the original literatures. The potential of our work is still largely untapped due to the enormous computational power necessary to train the necessary number of networks for obtaining the best network architectures through evolution. It is worth noting that solving the problem of meta-learning still benefits significantly from having more computational resources because of parallelism, and replicating the same level of results at a small scale is still nearly impossible, as our results did not reach the same level as the original literatures (CoDeepNEAT at 0.927 and hierarchical Search at 0.963 test accuracy). However, from implementing and experimenting with the algorithms, we found that CoDeepNEAT was effective at both speciation and evolving a population of good-performing architectures with variations; hierarchical search was effective at both encoding and simplifying complex architectures. As more computational resources and time become more available to us, we expect to evolve novel network architectures while still achieving, at the very least, the benchmark scores on the PTB and CIFAR-10 datasets.

7 Future Work

The most immediate next goal of this work will be to evolve, using both CoDeepNEAT and the hierarchical representation algorithms, and test DNNs for the task of modeling the PTB. Initial (but incomplete) experimentation has been promising, with validation-perplexity in the range of approximately 150-220 when with just a few epochs of training through both methods. We aspire to improve upon the test-perplexity score of 78, the best score found in literature that was achieved on PTB with evolutionary algorithms [10]

Our ultimate goal is to create a single unified evolutionary framework capable of evolving a deep neural network optimally designed for any language modeling and image classification task. In order to do this, we've focused our effort on using evolution to search solely for high performing network topologies. Current work is focused on combining the hierarchical representation of neural networks with the multi-level evolution scheme of CoDeepNEAT to allow for simultaneous evolution on three different hierarchical levels.

References

[1] Cifar-10 and cifar-100 datasets. https://www.cs.toronto.edu/ kriz/cifar.html.

[2] The NEAT users page. www.cs.ucf.edu/∼kstanley/neat.html. Accessed: 2014-05-01.

[3] Anonymous. Hierarchical representations for efficient architecture search. *International Conference on Learning Representations*, 2018.

[4] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. *ArXiv e-prints*, September 2014.

[5] F. Chollet. Xception: Deep Learning with Depthwise Separable Convolutions. *ArXiv e-prints*, October 2016.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[8] Y. Kim, Y. Jernite, D. Sontag, and Alexander M. Rush. Character-Aware Neural Language Models. *Association for the Advancement of Artificial Intelligence*, February 2016.

[9] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.*, 19(2):313–330, June 1993.

[10] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, and B. Hodjat. Evolving deep neural networks. *ArXiv e-prints*, March 2017.

[11] D. E. Moriarty and R. Miikkulainen. Hierarchical evolution of neural networks. In *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*, pages 428–433, May 1998.

[12] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. Le, and A. Kurakin. Large-Scale Evolution of Image Classifiers. *ArXiv e-prints*, March 2017.

[13] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. M. A. Patwary, Prabhat, and R. P. Adams. Scalable Bayesian Optimization Using Deep Neural Networks. *ArXiv e-prints*, February 2015.

[14] Kenneth Stanley. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21, 2004.

[15] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning Transferable Architectures for Scalable Image Recognition. *ArXiv e-prints*, July 2017.