Developing a Volunteer Computing Project to Evolve Convolutional Neural Networks and Their Hyperparameters

Travis Desell
Department of Computer Science
University of North Dakota
Grand Forks, North Dakota 58202
Email: tdesell@cs.und.edu

***Abstract*—This work presents improvements to a neuro-evolution algorithm called Evolutionary eXploration of Augmenting Convolutional Topologies (EXACT), which is capable of evolving the structure of convolutional neural networks (CNNs). While EXACT has multithreaded and parallel implementations, it has also been implemented as part of a volunteer computing project at the Citizen Science Grid to provide truly large scale computing resources through over 5,500 volunteered computers. Improvements include the development of a new mutation operator, which increased the evolution rate by over an order of magnitude and was also shown to be significantly more reliable in generating new CNNs than the traditional method. Further, EXACT has been extended with a simplex hyperparameter optimization (SHO) method which allows for the co-evolution of hyperparameters, simplifying the task of their selection while generating smaller CNNs with similar predictive ability to those generated with fixed hyperparameters. Lastly, the backpropagation method has been updated with batch normalization and dropout. Compared to previous work, which only achieved prediction rates of 98.32% on the MNIST handwritten digits testing data after 60,000 evolved CNNs, these new advances allowed EXACT to achieve prediction rates of 99.43% within only 12,500 evolved CNNs – rates which are comparable to some of the best human designed CNNs.**

I. Introduction

Convolutional Neural Networks (CNNs) have become a highly active area of research due to strong results in areas such as image classification [1], [2], video classification [3], sentence classification [4], and speech recognition [5], among others. Significant progress has been made in the design of CNNs, from the venerable LeNet 5 [2] to more recent large and deep networks such as AlexNet [1], VGGNet [6], GoogleNet [7] and ResNet [8]. However, less work has been made in the area of automated design of CNNs.

There exist a number of neuroevolution techniques capable of evolving the structure of feed forward and recurrent neural networks, such as NEAT [9], HyperNEAT [10], CoSyNE [11], as well as ant colony optimization based approaches [12], [13]. However, these have not yet been applied to CNNs due to the size and structure of CNNs, not to mention the significant amount of time required to train one.

Koutník *et al.*have published a work titled *Evolving Deep Unsupervised Convolutional Networks for Vision-Based Reinforcement Learning* [14], however in this work the structure of the CNN used was held fixed while only a small recurrent neural network controller (which takes output from the CNN) was evolved using the CoSyNE [11] algorithm. Zoph *et al.* [15] have a yet to be published work which uses a recurrent neural network trained with reinforcement learning to maximize the expected accuracy of generated architectures on a validation set of images. However, this approach is gradient based and generates CNNs by layer, with each layer having a fixed filter width and height, stride width and height, and number of filters.

Highlighting the cutting edge nature of this topic, recent related preprints have been submitted to `arXiv.org`. Xie *et al.* propose a Genetic CNN method which encodes CNNs as binary strings [16], however they only evolve structure of convolution operations between pooling layers, and keep the filter sizes fixed. Miikkulainen *et al.* have proposed CoDeepNEAT, which is also based on NEAT with each node acting as an entire layer, with the type of layer and hyperparameters being co-evolved [17]. However, connections within layers are fixed depending on their type without arbitrary connections. Real *et al.*, have also evolved image classifiers on the CIFAR-10 and CIFAR-100 datasets in work most close to this [18]. They also use a distributed algorithm to evolve progressively more complex CNNs through mutation operations, and handle conflicts in filter sizes by reshaping non-primary edges with zeroth order interpolation.

This work presents advances to the Evolutionary eXploration of Augmenting Convolutional Topologies (EXACT), which can evolve CNNs of arbtrary structure, filter and feature map size. Due to high computational demands, it has been implemented as part of the Citizen Science Grid[1], a Berkeley Open Infrastructure for Network Computing (BOINC) [19] volunteer computing project. Advances include an *add node* mutation operation which significantly improves evolution speed; simplex hyperparameter optimization (SHO), which co-evolves hyperparameters; and improvements to the backpropagation algorithm used. Using the MNIST handwritten digits dataset [20] as a benchmark, over 5,500 volunteered computers

[1]http://csgrid.org

were capable of training CNNs with 99.43% accuracy on the test data in under 12,500 evaluated CNNs – results competitive with some of the best human architected CNNs [21]. These results are a significant of improvement over prior work [22], [23], which required 60,000 evaluated CNNs to reach 98.32% accuracy on the test data.

II. Evolutionary Exploration of Augmenting Convolutional Topologies

The EXACT algorithm starts with the observation that any two feature maps of any size within a CNN can be connected by a convolution of size $conv_d = |out_d - in_d| + 1$, where out_d and in_d are the size of the output and input feature maps, respectively, and $conv_d$ is the size of the convolution in dimension d. The consequence of this observation is that the structure of a CNN can be evolved solely by determining the sizes of the feature maps and how they are connected. Instead of evolving the weights of individual neurons and how they are connected, as done in the NEAT [9] algorithm, the architecture of a CNN can be evolved in a similar fashion except on the level of how feature maps are connected, with additional operators to modify the feature map sizes. Whereas NEAT works on the level of neurons and weights, EXACT works on the level of feature maps (or *nodes*) and filters (or *edges*).

Due to the computational expense of training CNNs, EXACT has been designed with scalable distributed execution in mind. It uses an asynchronous evolution strategy, which has been shown by Desell *et al.* to allow scalability up to potentially millions of compute hosts in a manner independent of population size [24]. A master process manages a population of *CNN genomes* (the feature map sizes and how they are connected) along with their fitness (the minimized error after backpropagation on that CNN). Worker processes request CNN genomes to evaluate from the master, which generates them either through applying mutation operations to a randomly selected genome in the population (see Section II-B) or by selecting two parents and performing crossover to generate a child genome (see Section II-C). When that worker completes training the CNN, it reports the CNN along with its fitness back to the master, which will insert it into the population and remove the least fit genome if it would improve the population. This asynchronous approach has an additional benefit in that the evolved CNNs have different training times, and no worker need wait in the results of another to request another CNN to evaluate, *i.e.*, this approach automatically load balances itself.

Given the strong advances made by in the machine learning community for training deep CNNs and the sheer number of weights in large CNNs, attempts to try and simultaneously evolve the weights in the neural networks did not seem feasible. Instead, EXACT allows for any CNN training method to be plugged in to perform the fitness evaluation done by the workers. In this way, EXACT can benefit from further advances by the machine learning community and also make for an interesting platform to evaluate different neural network training algorithms.

A. Population Initialization

Generation of the initial population starts with first generating a *minimal CNN genome*, which consists solely of the input node, which is the size of the training images (plus padding if desired), and one output node per training class for a softmax output layer, with one edge connecting the input node to each output node. In this case of this work which uses the MNIST handwritten digits dataset, this is a 28x28 input node, and 10 1x1 output nodes. This is sent as the CNN genome for the first work request, and a copy of it is inserted into the population with ∞ as fitness, denoting that it had not been evaluated yet. Further work requests are fulfilled by taking a random member of the population (which will be initially just the minimal CNN genome), mutating it, inserting a copy of the mutation into the population with ∞ as fitness and sending that mutated CNN genome to the worker to evaluate. Once the population has reached a user specified population size through inserting newly generated mutations and results received by workers, work requests are fulfilled by either mutation or crossover, depending on a user specified crossover rate (*e.g.*, a 20% crossover rate will result in 80% mutation).

B. Mutation Operations

When a CNN genome is selected for mutation, a user specified number of the following mutations are performed. In testing, this was found to be beneficial to allow for greater variation in the CNNs generated. Each operator is selected with a user specified rate. Currently, the CNNs evolved do not utilize pooling layers, however modifying the size and type of pooling done by the feature maps is an area of future work.

The operations performed are similar to the NEAT algorithm, with the addition of operations to change the node size, as well as a new *add node* operator. Additionally, whereas NEAT only requires innovation numbers for new edges, EXACT requires innovation numbers for both new nodes and new edges. The master process keeps track of all node and edge innovations made, which is required to perform the crossover operation in linear time without a graph matching algorithm.

a) Disable Edge: This operation randomly selects an enabled edge in a CNN genome and disables it so that it is not used. The edge remains in the genome. As the *disable edge* operation can potentailly make an output node unreachable, after all mutation operations have been performed to generate a child CNN genome, if any output node is unreachable that CNN genome is discarded and a new child is generation by another attempt at mutation.

b) Enable Edge: If there are any disabled edges in the CNN genome, this operation selects a disabled edge at random and enables it.

c) Split Edge: This operation selects an enabled edge at random and disables it. It creates a new node (creating a new node innovation) and two new edges (creating two new edge innovations), and connects the input node of the split edge to the new node, and the new node to the output node of the split edge. The feature map size of the new node is set to $\frac{i_x+o_x}{2.0}$ by $\frac{i_y+o_y}{2.0}$, where i_d and o_d are the size of the input and output feature maps, respectively, in dimension d (*i.e.*, the size of the new node is halfway between the size of the input and output nodes). Further, the new node is given a depth value, $depth_{new} = \frac{depth_{output}+depth_{input}}{2.0}$, which is used by the *add edge* operation and to linearly perform forward and backward propagation without graph traversal.

d) Add Edge: This operation selects two nodes n_1 and n_2 within the CNN Genome at random, such that $depth_{n_1} < depth_{n_2}$ and such that there is not already an edge between those nodes in this CNN Genome, and then adds an edge from n_1 to n_2. This ensures that all edges generated feed forward (currently EXACT does not evolve recurrent CNNs). If an edge between n_1 and n_2 exists within the master's innovation list, that edge innovation is used, otherwise this creates a new edge innovation.

e) Change Node Size: This operation selects a node at random from within the CNN Genome and randomly increases or decreases its feature map size in both the x and y dimension. For this work, the potential size modifications used were [-2, -1, +1, +2].

f) Change Node Size X: This operation is the same as *change node size* except that it only changes the feature map size in the x dimension.

g) Change Node Size Y: This operation is the same as *change node size* except that it only changes the feature map size in the y dimension.

h) Add Node: This operation selects a random depth between 0 and 1, noninclusive. Given that the input node is always depth 0 and the output nodes are always depth 1, this splits the CNN in two. A new node is created, at that depth, and 1-5 edges are randomly generated to nodes with a lesser depth, and 1-5 edges are randomly generated to nodes with a greater depth. The node size is set to the average of the maximum input node size and minimum output node size.

C. Crossover

Crossover utilizes two hyperparameters, the *more fit parent crossover rate* and the *less fit parent crossover rate*. Two parent CNN genomes are selected, and the child CNN genome is generated from every edge that appears in both parents. Edges that only appear in the more fit parent are added randomly at the *more fit parent crossover rate*, and edges that only appear in the less fit parent are added randomly at the *less fit parent crossover rate*. Edges not added by either parent are also carried over into the child CNN genome, however they are set to disabled. Nodes are then added for each input and output of an edge. If the more fit parent has a node with the same innovation number, it is added from the more fit parent (*i.e.*, feature map sizes are inherited from the more fit parent if possible), and from the less fit parent otherwise.

D. Epigenetic Weight Initialization

While EXACT is independent of the method used to train CNNs, it does however present an interesting opportunity for weight initialization. As after the initial population is evaluated, child genomes are generated from one or two trained parent CNNs. The EXACT implementation optionally allows for weights of the parent CNN genomes to be carried over into child genomes, as *"epigenetic" weight initialization* – as these weights are a modification of how the genome is expressed as opposed to a modification of the genome itself.

III. Simplex Hyperparameter Optimization

Determining the best hyperparameters (the parameters used by backpropagation) is a challenging task, especially as the heuristics for deep learning grow more complicated. The parameters are interrelated and there can be a fairly large number of them. For example, in this work there are 11 different hyperparameters that can be modified to effect the training algorithm (see Section VI). How to best determine these hyperparameters is still an open problem, and can be very time consuming.

As the EXACT algorithm keeps a population of well performing CNNs, each potentially with their own hyperparameters, this also allows for the *co-evolution* of those hyperparameters. This work leverages a simple but effective asynchronous strategy based on the Nelder-Mead simplex [25] that has been shown by the author to be effective for global optimization problems [26]. This *simplex hyperparameter optimization* (SHO) selects N (in this work, $N = 5$) distinct individuals at random from the population. It uses the gradient between the hyperparameters of the best selected genome and the average of the hyperparameters of the others. Where r is a random point along the line bounded between l_1 and l_2 multiplied by the gradient (for this work $l_1 = 2.0$ and $l_2 = 0.5$), each of the i new hyperparameters, $h_{new,i}$, is calculated as a random point along the line between the average, $h_{avg,i}$ and best, $h_{best,i}$ hyperparameters:

$$r = (rand(0,1) * l_1) - l_2 \tag{1}$$

$$h_{new,i} = h_{avg,i} + r * (h_{best,i} - h_{avg,i}) \tag{2}$$

IV. Evolving Networks for Generalizability

In the previous work [22], [23], the fitness used to determine if a CNN was inserted into the popualtion was based on the training error, calculated as the cross entropy loss of the softmax layer. However, this does not really reflect how humans typically select hyperparameters and neural network architectures – they try different settings and compare how well the CNN performed on the testing data, modifying hyperparameters and architectures to minimize the test error and maximize the number of correct predictions on the test data. Further, when batch normalization and dropout were added to the backpropagation algorithm (see Section VI), it was easy to reach over 99.9% on the training data, however results on the test data were still relatively poor at 98.38%. The real objective is to optimize the *generalizability* of the CNNs, or how well they perform on the unseen test data.

However, there are some concerns with simply utilizing the test error as a fitness value – EXACT and the SHO could potentially "cherry pick" architectures and hyperparameters which perform well on the test data set, but may not be generalizable to other unseen data. This is especially a concern when the epigenetic weight initialization strategy is used, as this strategy preserves weights which perform well on the test data. To address this concern, the MNIST test data was split up into two separate data sets of 5,000. The first, called the *generalizability data* was generated by selecting 500 of each class at random from the test data. The error on the generalizability data was used as the fitness to select CNNs, and the remaining 5,000 images of test data were never seen by either the backpropagation algorithm or EXACT for fitness selection. In this way, it is possible to be sure that the evolved CNNs are really generalizable to unseen data by testing how performed on the separate test data.

V. EXACT on Volunteered Hosts

The EXACT source code has been made freely available as an open source project on GitHub[2]. It has a multithreaded implementation for small scale use, and an MPI implementation for use on high performance computing clusters. However, to provide enough computational resources to perform EXACT on a large scale, it was also implemented as part of a BOINC project. BOINC clients running on volunteered hosts serve as worker processes, and server side daemons were developed to validate results and handle the master EXACT process. This required developing code to save the state of the EXACT master process to a MySQL database such that these server side daemons could be stopped and restarted without loss of progress, which enabled checkpointing in the multithreaded and MPI versions as well. Due to space limitations, the author recommends [19], [27] for further details on the BOINC architecture.

In able to be able to utilize BOINC as a platform for training evolved CNNs, a number of implementation challenges needed to be addressed. First, BOINC requires applications to be written in C++ so they can link against the required BOINC libraries to be executed within by the BOINC clients running on volunteered hosts. While there is some work in enabling the use of Python applications in BOINC [28] or applications within virtual machines [29], these efforts are still under development. The requirement of C++, along with other techical requirements described later in this section, precluded the use of popular CNN training packages such as Caffe [30] or Theano [31], [32].

Perhaps the most significant techincal challenge was that in order to prevent users from reporting incorrect or malicious results, each *workunit* needed to be sent to multiple hosts so that the results can be validated against each other. Results which do not match those from other hosts are flagged as invalid and discarded. Only results that are valid award that particular user credit, which can be used to generate currency for Gridcoin whitelisted projects [33], or as motivation to climb various leaderboards [34]. The applications that run on the hosts must also be able to checkpoint and resume from previously saved states, as users expect to not lose progress if they need to temporarily turn off their BOINC client, or restart or shut down their computers.

[2]https://github.com/travisdesell/exact

For many applications, checkpointing and validation can be easily addressed, however in the case of training CNNs, even slightly different values can accumulate leading to dramatically different results. This means two hosts could be doing valid work, but return results that are incomparable. This also necessitates care being taken by checkpoints. If the CNN weights and other training values aren't saved and restored with the exact same precision, these small differences can result in significantly different final results.

A. Validation

The EXACT code was able to accomplish identical results on a wide variety of BOINC computing hosts. In the current EXACT codebase, Windows hosts from XP to Windows 10 on *Intel86* and *x86_64* architectures, along with Mac OS X and Linux hosts on *x86_64* architectures are all returning practically identical results, allowing validation. As the Windows applications were compiled with Visual Studio 2015, the Linux applications were compiled with the Gnu C++ Compiler (g++) and the Mac OS X applications were compiled with the LLVM compiler, inconsistent implementations of the C and C++ Standard Libraries needed to be addressed.

First, at the end of each epoch, the order of training images needs to be shuffled. While the *minstd_rand0* random number generator returned indentical results across platforms, the C++ *shuffle* function did not due to differing implementations of the *uniform_distribution* classes. Due to this, EXACT utilizes a custom built Fisher Yates Shuffle function which produces identical results across hosts. Second, on the different platforms, the *exp()* function began to return slightly different results for more extreme input values, resulting in widely varying final results. EXACT uses a custom built *exp()* based on a Taylor Series expansion, which while slower than the C Standard Library implementations, returns identical results across the different platforms. As the *exp()* function was only used 10 times per image as part of the softmax layer, this did not have any noticable effect on performance.

B. Checkpointing

Due to issues involved in robustly developing portable binary files, checkpointing of the CNNs in EXACT is done using a text file. However, similarly to the *exp()* function, printing double precision variables to the checkpoint file with arbitrary precision lead to some data loss and divergent results. This issue was solved by the use of *hexfloats*, which allow full precision I/O of floating point variables to and from text files. The *minstd_rand0* was also used to ensure correct checkpointing, as other more advanced random number generators in the C++ Standard Library, like *mt19337* (Mersenne Twister) and *minstd_rand* utilized different serialization implementations so it was not possible to utilize them to send a pre-seeded random number generator as part of a BOINC workunit or return result.

VI. Backpropagation Implementation

For this work, a fairly standard implementation of stochastic backpropagation was used to train the CNNs, however there were some notable modifications due to the fact that the CNNs were evolved arbitrarily, as opposed to having well defined layers as in most human architected CNNs. Backpropagation was done using batch normalization [35] with varying batch sizes and potentially dropout [36] (in some cases the input and hidden dropout probabilities were set to 0).

a) Weight Initialization: Apart from the softmax output layer, each node in the CNNs evolved by EXACT used a leaky ReLU activation function with max value 5.5 and a leak of 0.1. Due to this fact, weights are initialized as recommended by He *et al.* [37], where the variance, σ^2, of the weights, w, input to a neuron is $\sigma^2(w) = \sqrt{\frac{2}{n}}$, where n is the number of weights input to that neuron. After a CNN has been generated by EXACT, when it is initialized a forward pass is made through the graph, and each node calculates the total number of weights input to it, and that value is then used to randomly initialize those weights.

b) Forward Propagation: As specified in Section II-B, each node in the evolved CNN contains a depth value. By placing all edges into a vector sorted by the depth of their input node, a forward pass through the CNN can be done by propagating forward through each edge in that sorted order as opposed to doing a graph traversal. Each node keeps track of the number of input edges, and when that many have propagated values forward into the node, it applies the ReLU activation function to each of its neurons, unless it is an output node.

c) Backward Propagation: Error values can be propagated backwards by each edge in the reverse order of the forward propagation. Weights were updated using Nesterov momentum and L2 Regularization [38]:

$$v_{prev} = v \tag{3}$$
$$v = \mu * v - \eta * dw_i \tag{4}$$
$$w_i += -\mu * v_{prev} + (1 + \mu) * v \tag{5}$$
$$w_i -= w_i * \lambda \tag{6}$$

where v_{prev} is the previous value for velocity v, μ is the momentum hyperparameter, η is the learning rate, λ is the L2 regularization weight decay hyperparameter, dw_i is the weight update calculated by error backpropagation for weight w_i. Velocities for Nesterov momentum are reset after every ω training examples.

d) Hyperparameter Updates: At the end of each epoch, the following updates are performed on the three hyperparameters μ, η, and λ:

$$\mu = \mu_{max} - ((\mu_{max} - \mu) * \Delta\mu); \tag{7}$$
$$\eta = max(\eta * \Delta\eta, \eta_{min}) \tag{8}$$
$$\lambda = max(\lambda * \Delta\lambda, \lambda_{min}) \tag{9}$$

where $\Delta\eta < 1$, $\Delta\lambda < 1$, and $\Delta\mu < 1$. These operations decay or increase the hyperparameters to the predefined μ_{max}, η_{min} and λ_{min} values. These values have been set to $\mu_{max} = 0.99$, $eta_{min} = 0.00001$, and $\lambda_{min} = 0.000001$.

VII. Results

A. Previous Results

In previous work [22], [23], over a period of two months, approximately 4,500 volunteered computers on the Citizen Science Grid trained over 120,000 CNNs using the EXACT algorithm on two different searches. One used epigenetic weight initialization and the other used randomized weight resets. Table I shows the error and prediction rates for these two searches. These were trained with the following hyperparameters: momentum $\mu = 0.5$, $\Delta\mu = 0.90$, learning rate $\eta = 0.001$, $\Delta\eta = 0.98$, and weight decay $\lambda = 0.00001$, $\Delta\lambda = 0.98$. Velocities for Nesterov momentum were not reset. These were chosen in part because they provided the most consistently good results on a LeNet based benchmark network. The CNNs were initially allowed to train for 50 epochs, which was increased to 100 epochs and then finally 150 epochs as it became apparent that the CNNs could reach lower training error with more epochs. The population size for each EXACT search was set to 100.

There were some notable differences between these initial results and the ones presented in this work. First, neither dropout [36] nor batch normalization [35], nor a velocity reset were implemented in the backpropagation algorithm used. These updates to the training process alone raised the training error on the LeNet based benchmark used to test backpropagation from 96.79% to 98.38%, for an average of 10 runs with random initialization. As stated in Section IV, the training error was used for CNN selection by EXACT, as opposed to the generalizability error introduced in this work. Further, the *add node* mutation operator had not yet been utilized as part of the EXACT algorithm. Optimizations have also made to the BOINC application since then, involving linearizing and optimizing memory usage for node values and edge weights, as well as converting from double to single precision floating point operations which resulted in an approximately 2-3x speedup and 4x reduction of memory usage.

B. New Results

Due to the optimizations made since prior work, and an increase to approximately 5,500 volunteered computers, new results were gathered by running five searches simultaneously, in two different batches, the first batch of five searches using the simplex method for hyperparameter optimization, and the second batch using fixed hyperparameters. All searches utilized epigenetic weight initialization as this had shown better results in the prior work. Each search was allowed to run until 12,500 evolved CNNs were reported, and as batch normalization provided significantly faster training, each CNN was allowed to train for only 50 epochs. The population

Network	Avg. Num. Weights	Training Error			Testing Error			Training Predictions			Testing Predictions		
		Best	Avg	Worst	Best	Avg	Worst	Best	Avg	Worst	Best	Avg	Worst
Randomized	25,603.35	3,494.54	3,742.22	3,825.23	544.26	603.17	682.75	97.75%	97.33%	97.05%	97.89%	97.40%	96.98%
Epigenetic	23,862.65	3,644.30	3,909.88	3,991.73	594.13	657.48	710.25	98.42%	97.98%	97.48%	98.32%	97.87%	97.28%

TABLE I: Error and Prediction Rates for the top 20 CNNs in each EXACT search after 60,000 evaluated CNNs from prior results.

	Simplex 1	Simplex 2	Simplex 3	Simplex 4	Simplex 5		Fixed 1	Fixed 2	Fixed 3	Fixed 4	Fixed 5		Avg Simplex	Avg Fixed
Best Training Error	225.203	256.195	236.661	280.256	**180.473**		502.339	497.575	633.442	510.248	643.029		**235.758**	557.327
Best Gen. Error	108.824	116.136	103.601	111.593	103.080		113.949	**96.309**	125.429	106.989	122.527		**108.647**	113.041
Best Testing Error	95.629	101.524	90.160	98.443	**81.814**		96.034	83.767	101.341	94.021	98.473		**93.514**	94.727
Best Gen+Test Error	204.946	221.009	200.885	218.204	189.446		215.459	**188.443**	232.969	203.355	223.398		**206.898**	212.725
Avg. Training Error	394.542	447.053	461.509	449.518	**378.490**		698.013	604.125	834.981	683.894	825.316		**426.222**	729.266
Avg. Gen. Error	130.772	127.178	118.568	129.946	120.114		130.180	**114.463**	139.940	130.228	137.914		**125.316**	130.545
Avg. Testing Error	129.006	117.089	107.458	122.990	105.798		118.832	**105.585**	123.968	114.299	126.818		**116.468**	117.900
Avg. Gen+Test Error	259.778	244.267	226.026	252.937	225.913		249.013	**220.048**	263.908	244.528	264.731		**241.784**	248.446
Worst Training Error	624.734	756.249	738.220	672.943	779.987		1,050.524	845.456	1,105.528	900.335	**1,028.785**		**714.427**	986.126
Worst Gen. Error	140.177	134.631	127.989	138.404	129.600		141.354	**122.931**	146.202	137.209	145.532		**134.160**	138.646
Worst Testing Error	154.564	139.170	**131.315**	154.928	144.082		146.091	132.534	150.923	147.623	147.290		**144.812**	144.892
Worst Gen+Test Error	292.775	267.813	257.145	288.679	271.466		284.485	**248.345**	293.409	282.729	290.398		**275.576**	279.873

TABLE II: Evolved Neural Network Error Rates

	Simplex 1	Simplex 2	Simplex 3	Simplex 4	Simplex 5		Fixed 1	Fixed 2	Fixed 3	Fixed 4	Fixed 5		Avg Simplex	Avg Fixed
Best Training Pred	99.978	99.968	99.975	99.972	**99.982**		99.877	99.903	99.845	99.903	99.838		**99.975**	99.873
Best Gen. Pred	99.340	99.360	99.360	99.380	99.360		99.400	**99.420**	99.260	99.320	99.320		**99.360**	99.344
Best Testing Pred	99.340	99.400	99.460	99.460	**99.520**		99.440	**99.520**	99.400	99.500	99.420		99.436	**99.456**
Best Gen+Test Pred	99.260	99.370	99.360	99.340	99.420		99.400	**99.430**	99.300	99.410	99.340		99.350	**99.376**
Avg. Training Pred	**99.951**	99.917	99.926	99.927	99.938		99.813	99.846	99.745	99.816	99.749		**99.932**	99.794
Avg. Gen. Pred	99.134	99.178	99.180	99.169	99.167		99.154	**99.223**	99.077	99.146	99.091		**99.166**	99.138
Avg. Testing Pred	99.146	99.252	99.286	99.201	**99.308**		99.229	**99.308**	99.215	99.270	99.207		99.239	**99.246**
Avg. Gen+Test Pred	99.140	99.215	99.233	99.185	99.237		99.191	**99.266**	99.146	99.208	99.149		**99.202**	99.192
Worst Training Pred	**99.890**	99.805	99.825	99.843	99.793		99.628	99.748	99.620	99.698	99.643		**99.831**	99.667
Worst Gen. Pred	98.960	98.980	**99.000**	98.960	98.980		98.920	98.980	98.840	98.940	98.860		**98.976**	98.908
Worst Testing Pred	98.960	98.980	**99.080**	99.040	99.080		98.980	99.000	98.920	99.040	98.920		**99.028**	98.972
Worst Gen+Test Pred	99.010	99.070	**99.090**	99.010	99.070		99.000	99.080	98.970	99.050	98.950		**99.050**	99.010

TABLE III: Evolved Neural Network Prediction Rates

size was also reduced to 50. Each batch of searches required approximately 2 weeks. Tables II and III present the resulting error and prediction rates, respectively, of the simplex and fixed hyperparameter searches. Tables IV and V present the resulting number of weights and edges, respectively, of the CNNs in each search.

1) Initial Hyperparameters: The simplex hyperparameter optimization started with the following hyperparameter ranges: momentum $\mu = 0.4$ to 0.6, $\Delta\mu = 0.90$ to 0.99, learning rate $\eta = 0.001$ to 0.03, $\Delta\eta = 0.90$ to 0.99, and weight decay $\lambda = 0.001$ to 0.0001, $\Delta\lambda = 0.90$ to 0.99, a velocity reset $\omega = 500$ to 3000, an input dropout probability of 0.0005 to 0.002, a hidden dropout probability of 0.05 to 0.15, a batch size of 25 to 150, and a batch normalization alpha $\alpha = 0.001$ to 0.2. After allowing the hyperparemters to initialize after 500 CNNs were reported, these constraints were then relaxed to: momentum $\mu = 0.0$ to 0.99, $\Delta\mu = 0.0$ to 1.0, learning rate $\eta = 0.00001$ to 0.1, $\Delta\eta = 0.00000001$ to 1.0, and weight decay $\lambda = 0.0$ to 0.1, $\Delta\lambda = 0.00000001$ to 1.0, a velocity reset $\omega = 0$ to $60,000$, an input dropout probability of 0.0001 to 0.5, a hidden dropout probability of 0.0 to 0.9, a batch size of 25 to 300, and a batch normalization alpha $\alpha = 0.0001$ to 0.5.

The fixed parameter searches were trained with the following hyperparameters: momentum $\mu = 0.5$, $\Delta\mu = 0.95$, learning rate $\eta = 0.0025$, $\Delta\eta = 0.95$, and weight decay $\lambda = 0.0005$, $\Delta\lambda = 0.95$, a velocity reset $\omega = 1000$, an input dropout probability of 0.0, a hidden dropout probability of 0.0, a batch size of 50, and a batch normalization alpha $\alpha = 0.1$. These were used as they were close to the average of what the simplex hyperparameter optimization had evolved to, and the dropout probabilities were set to 0 due to the suggestion in the batch normalization paper that it generally makes dropout unnecessary on MNIST [35].

2) Prediction, Error and CNN Size: Over previous work, both the ability for the EXACT search to evolve complex architectures along with their predictive ability have been significantly increased. In previous work, it took 60,000 evolved CNNs to reach an average population sizes of 25,603 and 23,862 weights, while the new batches of searches reached 59,474 and 64,167 weights (as an average of the search populations) in only 12,500 evolved CNNs. The best predictions on all 10,000 testing images increased from 98.32% and 97.89% to 99.43% and 99.42%, making these competitive with some of the best human architected CNNs [21].

In comparing the searches with SHO vs. the fixed hyperparameters, in general the training error on average was half as much for the simplex vs. the fixed hyperparameters, however only a slight improvement was seen on the generalizability, testing, and combined error rates. This resulted in close to identical prediction rates for both sets of searches (99.35% vs. 99.376% on average on the combined error rates). However, there is one notable difference between the two hyperparmeter methods. As whole, on average the simplex searches had approximately 5,000 (7.3%) less weights and 11 (2.3%) less edges. Further, for the best CNNs found, they had approxi-

	Simplex 1	Simplex 2	Simplex 3	Simplex 4	Simplex 5		Fixed 1	Fixed 2	Fixed 3	Fixed 4	Fixed 5		Avg Simplex	Avg Fixed
Average	53,398.468	65,713.055	61,095.377	54,203.910	62,964.051		73,687.986	75,504.441	53,093.571	66,739.344	**51,812.990**		**59,474.972**	64,167.666
Best Training Error	61399	70773	62868	59969	75893		94444	87261	58534	80025	**56285**		**66,180.400**	75,309.800
Best Gen Error	61495	74024	63301	61955	73339		79195	85114	64727	81779	**57011**		**66,822.800**	73,565.200
Best Test Error	63190	74838	66532	55876	77783		93754	74886	57874	83081	**52500**		**67,643.800**	72,419.000
Best Gen+Test Error	63190	74839	66532	56573	77783		93754	85114	57342	81779	**52500**		**67,783.400**	74,097.800
Worst Training Error	51216	53320	51829	49114	38213		47322	55871	**37961**	54050	48114		48,738.400	**48,663.600**
Worst Test Error	48217	53320	47560	45690	**38361**		47322	66655	46121	44928	44403		**46,629.600**	49,885.800
Worst Gen Error	56370	68703	54403	56944	72589		82512	64768	**52248**	70916	56109		**61,801.800**	65,310.600
Worst Gen+Test Error	40474	52747	47560	45690	**38361**		78812	56373	51062	44928	46628		**44,966.400**	55,560.600

TABLE IV: Evolved Neural Network Weight Counts

	Simplex 1	Simplex 2	Simplex 3	Simplex 4	Simplex 5		Fixed 1	Fixed 2	Fixed 3	Fixed 4	Fixed 5		Avg Simplex	Avg Fixed
Average	496.415	496.151	461.333	409.488	486.775		498.620	630.456	**364.046**	471.957	441.522		**470.032**	481.320
Best Training Error	616	536	485	477	631		667	770	**426**	578	486		**549.000**	585.400
Best Gen Error	600	578	**484**	488	593		559	738	490	625	500		**548.600**	582.400
Best Test Error	635	589	506	430	649		656	617	**420**	625	447		561.800	**553.000**
Best Gen+Test Error	635	589	506	439	649		656	738	**403**	625	447		**563.600**	573.800
Worst Training Error	467	368	369	354	265		284	410	**242**	326	405		364.600	**333.400**
Worst Gen Error	536	515	373	447	578		570	505	**341**	512	486		489.800	**482.800**
Worst Test Error	406	368	330	323	**266**		284	544	300	298	365		**338.600**	358.200
Worst Gen+Test Error	344	360	330	323	**266**		546	444	330	298	384		**324.600**	400.400

TABLE V: Evolved Neural Network Edge Counts

mately 9,000 (12%) less weights and 36 (6.2%) less edges, meaning that the simplex hyperparameter optimziation was able to train more efficient CNNs, finding well performing CNNs more quickly.

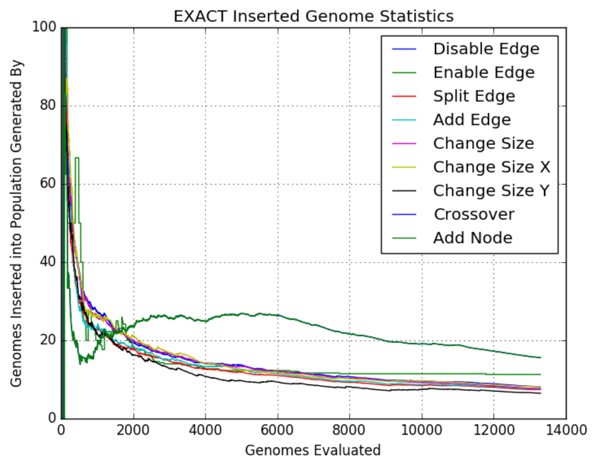
These results are rather interesting, given the difference in training error, however perhaps completely not expected due to the fact that the fixed hyperparameters were based on the best found hyperparameters of the simplex hyperparameter optimization. Further, given the fact that weights were reused by epigenetic weight initialization, the simplex hyperparameter optimization had an opportunity to allow the CNNs to utilize hyperparameters that were more refined to how far the weights had already been trained. A possible reason for the similar combined prediction rates may be that the evolved neural networks are reaching the limit of the predictive ability of the architecture due to the lack of pooling layers, and/or the limits of the backpropagation heuristics utilized. This does make some sense as while the simplex hyperparamters were able to perform better on the training data, they were unable to reach any better results on the combined generalizability and test data.

3) Reproduction Operator Performance: Figure 1 shows how many CNNs were inserted into the population over time that were produced by crossover and the various mutation operators. Each CNN tracks which operations it was generated by (*e.g.*, 1 add node, 1 edge disable, 1 split edge). When any genome is reported, the total number of operations performed for each type is incremented by these values, and if it was inserted into the population, the total number of operations inserted for each type is also incremented. This allows the search to track how often a generated genome is inserted into the population based on what reproduction operations produced it. These results show that the new *add node* operation provides significant improvement to the population evolution, as CNNs generated by this operation are inserted into the population significantly more often than the standard operations based on those used by NEAT, especially in the case of fixed hyperparameters. The difference is most likely due to variation in the simplex generated hyperparameters leading to more fluctuation in training effectiveness. These results also open up an interesting question in that the neuroevolution operations standardly used by NEAT may be quite inefficient.

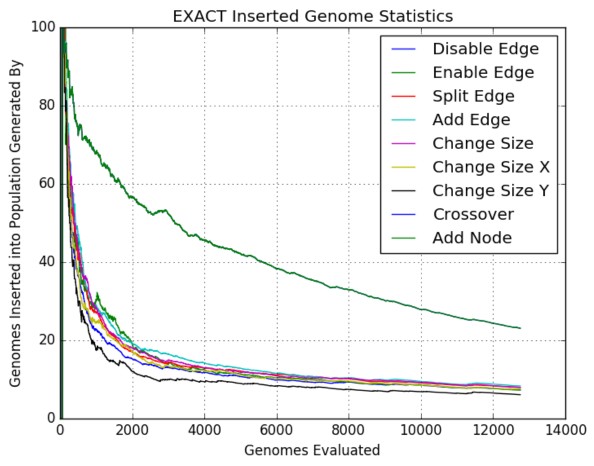
4) Hyperparameter Co-Evolution: Figure 2 presents statisics with regards to what hyperparameters were used to train the CNNs that were present in the population. These results show that the simplex hyperparameter optimization method allowed the hyperparamters in use to evolve over time and in some cases converge closely (*e.g.*, the learning rate). What is quite interesting however, is how many of the hyperparameters showed a wide range of values for the best individual in the population (the green lines). In many cases these appear bimodal, alternating between high and low values within the population. This may suggest that some sets of hyperparameters are being well used to find new different optima, while others are used to refine results within an already found optima of weights. This may also be part of the reason why the simplex searches had much better performance reducing training error, as some hyperparameters allowed more refinement of weights.

C. Evolved Genomes

Figure 3 show the best CNNs evolved by the simplex and fixed hyperparameter searches. These networks are quite interesting in that they are quite different from the highly structured CNNs found seen in literature [1], [2], [6]–[8]. It is interesting that there appear to be some nodes which have larger numbers of input and output edges, which may be of higher significance than others. Additionally, it should be noted that the edges directly from the input to the output were preserved, and many edges skip forward past multiple other nodes, features which bear some similarity to recent work in ResNets [8], [39].



(a) Reproduction operator insert rates for the simplex hyperparameter search with the best found CNN (simplex search 5).



(b) Reproduction operator insert rates for the fixed hyperparameter search with the best found CNN (fixed search 2).

Fig. 1: The rate which CNNs were inserted into the population based on how they were generated.

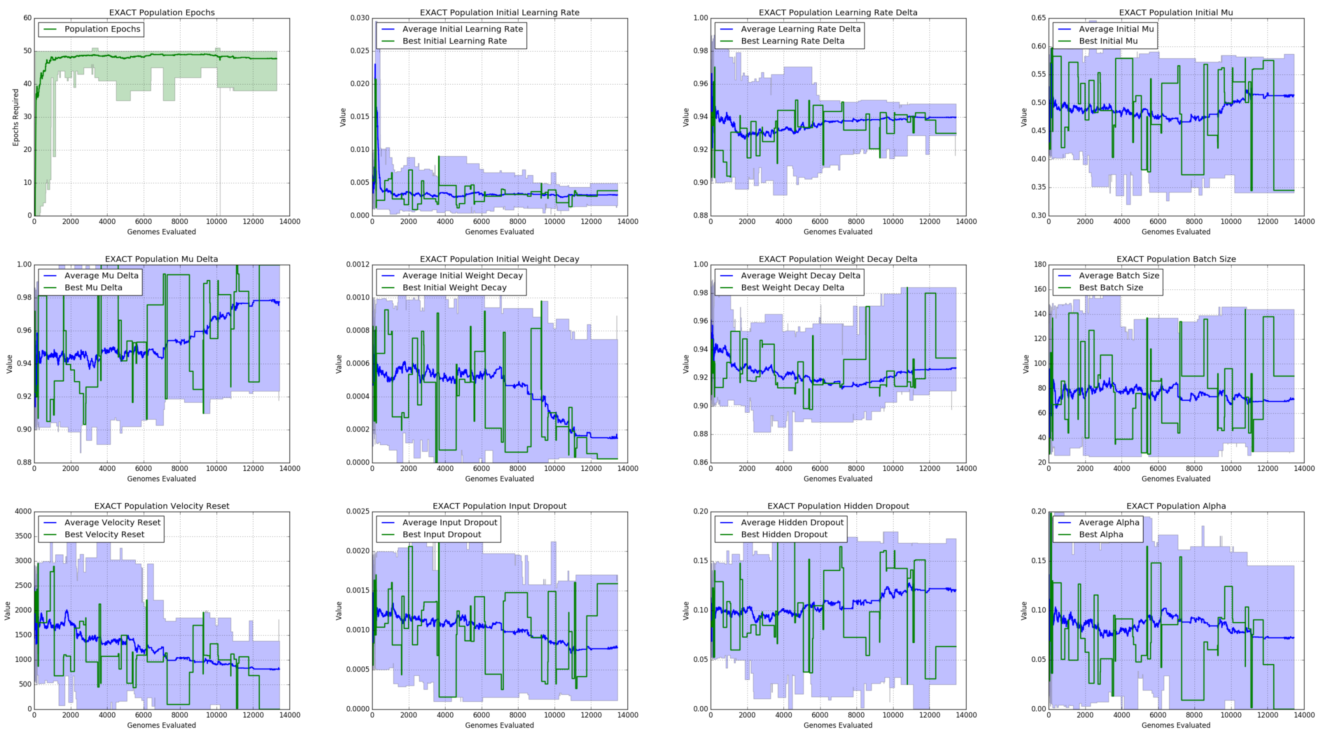
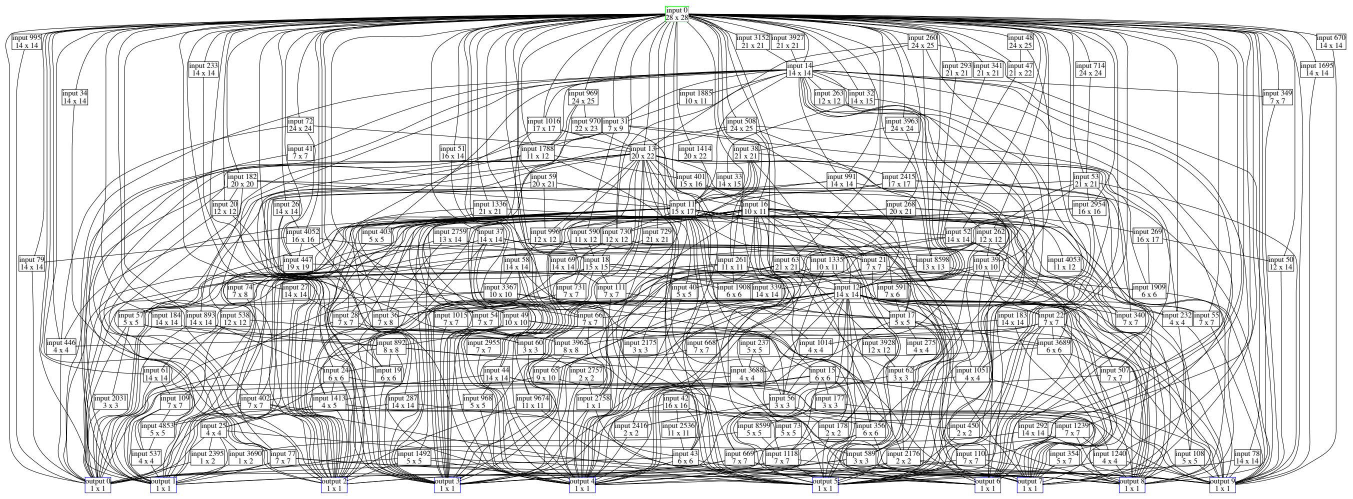


Fig. 2: Population statistics for number of epochs and the hyperparameter values for the simplex hyperparameter optimization method on the search which resulted in the best combined generalizability and test prediction rate (simplex search 5). The population epochs figure shows statistics for which epoch the CNNs found the best performing weights.


(a) The 11597th genome inserted into fixed search 2. This network had 630 edges and 76065 weights, and a combined generalizability and testing rate of 99.43%.

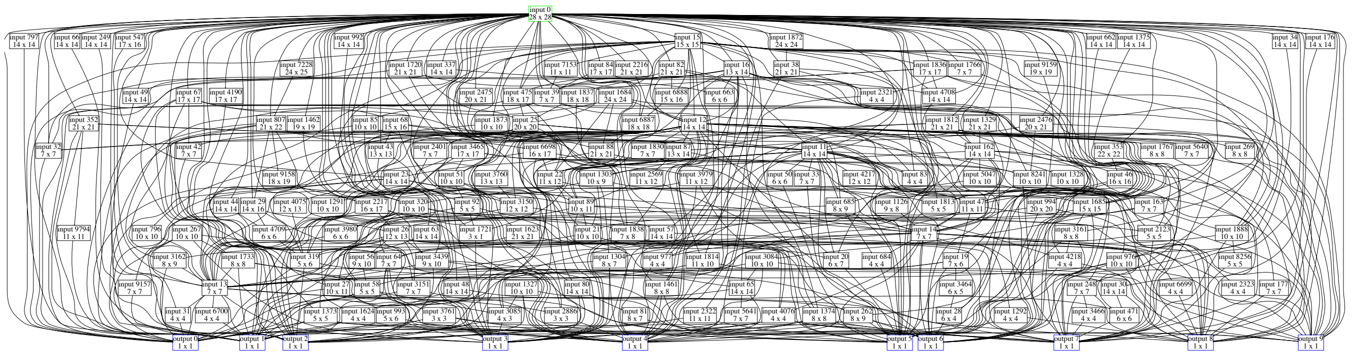

(b) The 12455th genome inserted into simplex search 5. This network had 610 edges and 73760 weights, and a combined generalizability and testing rate of 99.42%.

Fig. 3: The best performing CNNs evolved by the simplex and fixed hyperparameter searches. Disabled edges and unused nodes are not shown. The green square node at the top is the input node, and the 10 blue square nodes at the bottom represent each of the 10 outputs (for the numbers 0 – 9 in the MNIST dataset).

VIII. Discussion and Future Work

This work presents significant advancements to the performance of the EXACT algorithm: the addition of an add node mutation operator, simplex hyperparameter optimization (SHO), and improved backpropagation heuristics through dropout and batch normalization. This enabled the algorithm to evolve CNNs with 99.43% test accuracy in under 12,500 evalauted CNNs at 50 epochs per CNN, a dramatic improvement over the previous implementation which required 60,000 evaluated CNNs to reach only 98.38% test accuracy with 150 epochs per CNN. Further, SHO alleviates the challenge of determining backpropagation hyperparameters, while at the same time evolving smaller CNNs with similar predictive ability.

This work opens the door for significant future work. In particular, the EXACT algorithm does not yet evolve pooling layers, due to the fact that a max pooling layer of size 2 reduces the size of a feature map by a factor of two – which can in many cases result in the feature map being connected to large output feature maps. However, a potential way for this issue to be resolved is by utilizing fractional max pooling [40], which allows for fractional sized max pooling layers which can more easily be integrated into EXACT's arbirarily generated CNNs and allow even greater predictive ability.

Using EXACT to evolve CNNs for other data sets such as the CIFAR and TinyImage datasets [41], [42] is of high interest as well. SHO and the volunteer computing implementation provide significant means for the analysis of the effect of CNN architecture and hyperparameters on their generalizability, as thousands of CNNs are trained with varying hyperparameters and stored in a database. This gives an opportunity to utilize data mining strategies to obtain new insights on how CNNs function.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

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

[3] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725–1732.

[4] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.

[5] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

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

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

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

[10] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, "A hypercube-based encoding for evolving large-scale neural networks," *Artificial life*, vol. 15, no. 2, pp. 185–212, 2009.

[11] F. Gomez, J. Schmidhuber, and R. Miikkulainen, "Accelerated neural evolution through cooperatively coevolved synapses," *Journal of Machine Learning Research*, vol. 9, no. May, pp. 937–965, 2008.

[12] T. Desell, S. Clachar, J. Higgins, and B. Wild, "Evolving deep recurrent neural networks using ant colony optimization," in *Evolutionary Computation in Combinatorial Optimization*, ser. Lecture Notes in Computer Science, G. Ochoa and F. Chicano, Eds. Springer International Publishing, 2015, vol. 9026, pp. 86–98. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-16468-7_8

[13] K. Salama and A. M. Abdelbar, "A novel ant colony algorithm for building neural network topologies," in *Swarm Intelligence*. Springer, 2014, pp. 1–12.

[14] J. Koutník, J. Schmidhuber, and F. Gomez, "Evolving deep unsupervised convolutional networks for vision-based reinforcement learning," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2014, pp. 541–548.

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

[16] L. Xie and A. Yuille, "Genetic cnn," *arXiv preprint arXiv:1703.01513*, 2017.

[17] 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 preprint arXiv:1703.00548*, 2017.

[18] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, Q. Le, and A. Kurakin, "Large-scale evolution of image classifiers," *arXiv preprint arXiv:1703.01041*, 2017.

[19] D. P. Anderson, E. Korpela, and R. Walton, "High-performance task distribution for volunteer computing." in *e-Science*. IEEE Computer Society, 2005, pp. 196–203.

[20] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits," 1998.

[21] R. Benenson, "Who is the best at mnist?" [Accessed Online 2017] http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html.

[22] T. Desell, "Large scale evolution of convolutional neural networks using volunteer computing," in *The Genetic and Evolutionary Computation Conference (GECCO)*, July 2017, pp. 2–pp, to Appear.

[23] ——, "Large scale evolution of convolutional neural networks using volunteer computing," *CoRR*, vol. abs/1703.05422, pp. 17–pp, 2017. [Online]. Available: http://arxiv.org/abs/1703.05422

[24] T. Desell, D. Anderson, M. Magdon-Ismail, B. S. Heidi Newberg, and C. Varela, "An analysis of massively distributed evolutionary algorithms," in *The 2010 IEEE congress on evolutionary computation (IEEE CEC 2010)*, Barcelona, Spain, July 2010.

[25] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The computer journal*, vol. 7, no. 4, pp. 308–313, 1965.

[26] T. Desell, C. Varela, and B. Szymanski, "An asynchronous hybrid genetic-simplex search for modeling the Milky Way galaxy using volunteer computing," in *Genetic and Evolutionary Computation Conference (GECCO)*, Atlanta, Georgia, July 2008.

[27] D. P. Anderson, "Volunteer computing: the ultimate cloud," *Crossroads*, vol. 16, no. 3, pp. 7–10, 2010.

[28] E. M. Heien, Y. Takata, K. Hagihara, and A. Kornafeld, "Pymw-a python module for desktop grid and volunteer computing," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–7.

[29] J. Rantala, "VMWrapper," 2017, http://boinc.berkeley.edu/trac/wiki/VmApps.

[30] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the ACM International Conference on Multimedia*. ACM, 2014, pp. 675–678.

[31] J. Bergstra, F. Bastien, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, D. Warde-Farley, I. Goodfellow, A. Bergeron *et al.*, "Theano: Deep learning on gpus with python," in *NIPS 2011, BigLearning Workshop, Granada, Spain*, vol. 3. Citeseer, 2011.

[32] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio, "Theano: new features and speed improvements," *arXiv preprint arXiv:1211.5590*, 2012.

[33] Gridcoin, "What is gridcoin?" 2017, http://www.gridcoin.us/.

[34] BoincStats, "BOINC stats," 2017, http://boincstats.com/.

[35] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, 2015, pp. 448–456.

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

[37] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[38] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu, "Advances in optimizing recurrent networks," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 8624–8628.

[39] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning." in *AAAI*, 2017, pp. 4278–4284.

[40] B. Graham, "Fractional max-pooling," *arXiv preprint arXiv:1412.6071*, 2014.

[41] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," *Computer Science Department, University of Toronto, Tech. Rep*, 2009.

[42] A. Torralba, R. Fergus, and W. T. Freeman, "80 million tiny images: A large data set for nonparametric object and scene recognition," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 30, no. 11, pp. 1958–1970, 2008.