EvoNN - A Customizable Evolutionary Neural Network with Heterogenous Activation Functions

Boris Shabash
Simon Fraser University
School of Computing Science
Burnaby, BC
bshabash@sfu.ca

Kay C. Wiese*
Simon Fraser University
School of Computing Science
Burnaby, BC
wiese@sfu.ca

ABSTRACT

While some attention has been given to the process of using Evolutionary Computation (EC) to optimize the activation functions within hidden layers, available activation function sets have always been hard coded by the developers, and were immutable by users.

In this paper, we present EvoNN. While many other Neuro-evolution based tools or algorithms focus primarily on the evolution of either Neural Network (NN) architecture, or its weights, EvoNN focuses on simultaneous evolution of weights and the activation functions within hidden layers. The main novely offered by EvoNN lies in that users can provide additional activation functions to the EvoNN system to be employed as part of the "alphabet" of available functions. This feature gives users a greater degree of flexibility over which functions the evolutionary optimizer can utilize.

We employ a set of three test cases where we compare EvoNN to a standard NN, and observe encouraging results showing a superior performance of the EvoNN system. We also observe this increase in performance comes at the cost of additional run time, but note that for some applications, this can be a worthwhile trade-off.

CCS CONCEPTS

• **Computing methodologies → Neural networks**; **Genetic algorithms**;

KEYWORDS

Artificial Neural Networks, Evolutionary Computation, Fitness Functions, Evolving Neural Network Activation Functions

ACM Reference Format:
Boris Shabash and Kay C. Wiese. 2018. EvoNN - A Customizable Evolutionary Neural Network with Heterogenous Activation Functions. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan,* Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3205651.3208282

*Corresponding Author

GECCO '18 Companion, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-1-4503-5764-7/18/07…$15.00
https://doi.org/10.1145/3205651.3208282

1 INTRODUCTION

Neuro-evolution is the process of optimizing the weights, architecture, activation functions, or any combination thereof of Artificial Neural Networks (ANNs) [8]. ANNs [10] are machine learning constructs which are loosely based on the human mind architecture. ANNs are, generally speaking, composed of layers of neurons, where neurons can only transmit information between layers, but not within layers (Figure 1).

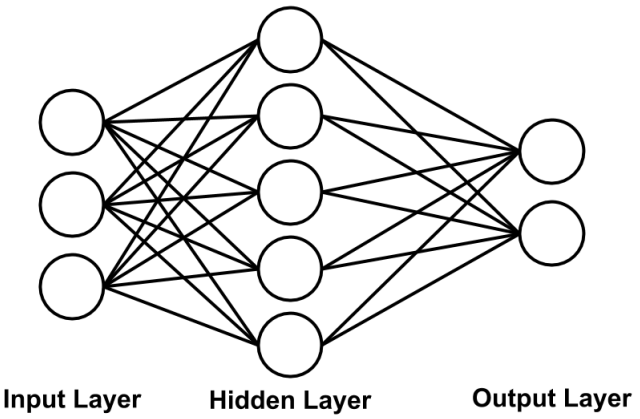


Figure 1: The general architecture of a neural network. This neural network contains 3 input neurons (thus able to process data with 3 features), 4 hidden neurons, and 2 output neurons (thus able to output a target of 2 values)

The most simple ANN contains three layers, and is composed of an input layer, a hidden layer, and an output layer, where each layer contains neurons. The neurons in each layer contain links to the preceding layer (except for the input layer neurons) and to the succeeding layer (except for the output layer neurons).

ANNs are composed of two major components, the weights on the links between the neurons, and the activation functions of neurons. Traditional methods of training neural networks, whether using backpropagation, or heuristic based training [18], have focused on adjusting the weights of the neural network [17, 20, 27], or the network's architecture [2, 5, 12, 23, 24], but relatively little work has been done on modifying the activation functions within neurons, with only a few notable exceptions. This is despite the fact that the choice of the transfer functions is as important as choice of architecture [7]

In [16], Mani suggests the modification of activation functions as part of the backpropagation algorithm, but his work can only be applied on a set of functions which can be ordered on a gradient. The work done by Liu and Yao in [14] focused on evolving both network architecture and the activation functions employed within the hidden nodes, but the nodes' activation functions were limited to either sigmoid or Gaussian activation functions. Alvarez, in [3], also worked on evolutionary development of activation functions, but employed a process similar to genetic programming [13] to evolve simple expressions to be utilized within both hidden and output neurons. Similarly, Augusteijn and Harrington [4] employed genetic programming activation functions as well with a set of predefined simple functions (such as tan(x), sqrt(x)) as the alphabet for the tree. An interesting approach to evolving activation functions was pursued by Turner in [25], where he evolved the connectivity of a predefined set of nodes in order to create the ANN. However, the resulting ANNs do not have a clear division into layers since the connectivity can occur between any two nodes. Similar work done by Stanley et al. [22], with the introduction of HyperNEAT, also allowed nodes to select and evolve their activation functions, and the work was more advanced than many other such attempts. However, the functions could still only be selected from a predefined set which could not be modified by the user. In 2017, Vasconcellos et al. introduced SUNA in [26]. Their work focused mostly on implementing an advanced neuron representation for neuroevolution, and reducing the number of neuroevolution hyper-parameters, and their advanced neurons also utilized variable functions. Yet similar to other methods, the activation function space was limited, and could not be extended by the user. Recently, in [21] Shirakawa et al. have also experimented with the simultaneous evolution of weights and activation functions, along with the evolution of drop out rate and other hyper-parameters in Deep Neural Networks (DNN). Their work produced very good results compared to using a predefined activation function. However, their choice for activation functions were limited to the ReLU and tanh activation functions, and did not allow for user interaction.

Though these examples showed promising results for ANNs with heterogeneous activation functions within layers, most research on such ANNs has focused on simple activation functions (with the exception of [25]), and none have focused on user supplied activation functions. Generally, this has been motivated by one of two reasons: First, while many programming languages allow for the run-time modification of variable values, the ability to use functions as variables and modify their value with ease is a relatively recent feature that came with high level programming languages such as Python and R. Second, there is a non-trivial run-time overhead added when modifying and assigning functions as variables. Since Machine Learning (ML) is usually concerned with making sense of large amounts of data, the minimization of run-time performance is incredibly important to the field.

In this work, we present EvoNN, an Evolutionary system for Neural Network evolution which is set apart by its ability to accept user specified activation functions, and demonstrate that while a system which includes function evolution, as well as weight evolution, performs slower, it can achieve superior performance due to its added flexibility.

2 METHODS

2.1 The Architecture of EvoNN

The basic unit of the ANN is the neuron, which is composed of two components: the weighted links leading information into the neuron, and the activation function. These components are summed up by the following formulas:

$$x = \sum_{i=0}^{N} w_{ij} a_i \tag{1}$$

$$a_{i+1} = f(x) \tag{2}$$

Where N is the number of neurons in the previous layer, a_i is the output of the i^{th} neuron in the previous layer, and w_{ij} is the weight of the link from the i^{th} neuron in the previous layer to the j^{th} neuron in the current layer. The activation function $f()$ is then applied to x to produce the current neuron's output, a_{i+1}.
Figure 2 illustrates this process.

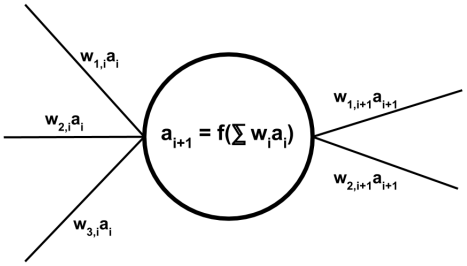


Figure 2: A sample neuron which accepts 3 inputs, applies an activation function $f()$ on their sum, and outputs a value which is then multiplied by the weights of the different outgoing edges.

The most commonly used activation function is the sigmoid function:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

but the hyperbolic tangent function is often used as well:

$$f(x) = \frac{2}{1 + e^{-x}} - 1 \tag{4}$$

Conventionally, the collection of neurons in a given layer all contain the same activation function. However, EvoNN allows different neurons within the same layer to have different activation functions (Figure 3). The function of each neuron becomes another parameter to be optimized. Figure 4 illustrates this framework.

The proposed framework has two fundamental advantages: Firstly, it presents a general added degree of flexibility for the neural network to take advantage of: While conventional neural networks force all hidden units to perform the same fundamental transformation to incoming data, the EvoNN framework allows each unit to respond differently (Notice however, it is not *obligated* to respond differently, since the optimizers can select for networks with the same activation function in all nodes). Second, since the specific framework of EvoNN does not use gradient-descent to optimize its networks, the functions made available to the optimizer *do not*

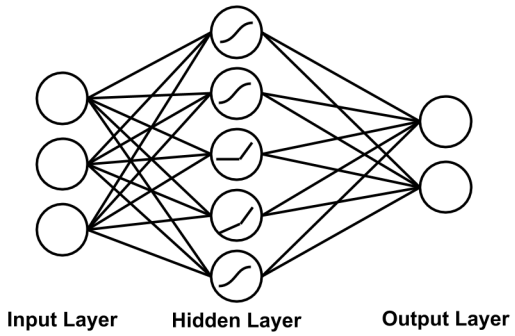
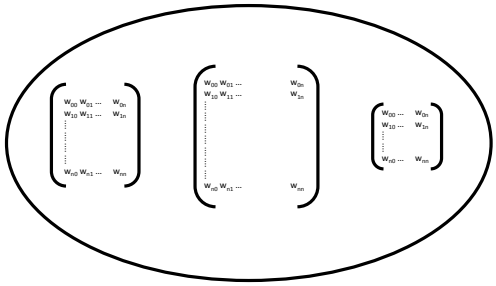
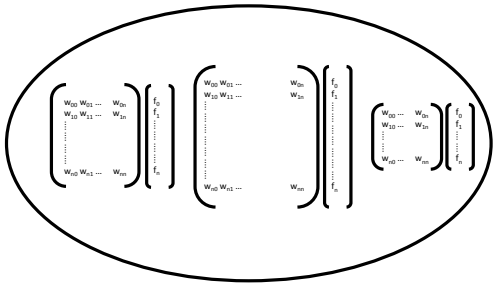


Figure 3: A sample neural network where the hidden layer contains the sigmoid activation function at the first, second, and fifth neorons, and piece-wise functions at the third and fourth neurons



(a) Generally, the chromosome of each individual using neuroevolution contains only the weights to be optimized, while the activation functions remain constants.



(b) In EvoNN, the chromosome of each individual contains an additional vector for each hidden layer which contains the activation function for each layer as well.

Figure 4: A comparison of chromosomes between standard neuroevolution frameworks, and individuals in the EvoNN framework. The example shows a network with three hidden layers

have to be differentiable, nor does their derivative need to be known ahead of time. The result of such an architecture is that users are able to supply EvoNN optimizer with their own functions (which have to conform to a particular signature), and EvoNN uses those functions within the optimization process.

2.1.1 Mutation. In the context of EvoNN mutation is divided into two types of modifications, each controlled by different parameters; Weight mutation is the process of modifying the weight associated with a specific link between two nodes. While activation function mutation modifies the activation function present in a node.

Weight mutation is controlled by the mutation probability ($P_m \in [0.0, 1.0]$) and the mutation radius ($R_m \in (0.0, \infty)$). The mutation probability determines the probability for each link to be mutated. While the mutation radius dictates the severity of the mutation. For example, if $P_m = 0.5$ and $R_m = 0.1$, then each link has a 50% chance of being mutated at each iteration. If the link is indeed chosen for mutation, its weight will increase by a value uniformly chosen from the range $[-0.1, 0.1]$. Notice this means the weight may decrease if the chosen value falls between -0.1 and 0.0.

Activation function mutation works similarly to weight mutation and is controlled by the function mutation probability ($P_{mf} \in [0.0, 1.0]$). Much like the weight mutation probability, the function mutation probability controls the likelihood an activation function within a hidden node is modified. However, this mutation does not have a mutation radius equivalent. If a node's activation function is chosen for mutation, the new function is uniformly chosen from the "bank" of available functions which are different from the function employed at the given node. For example, if $P_{mf} = 0.5$, then each hidden node has a 50% chance of having its activation functions mutated. If a node which contains the function $f(x)$ is chosen for the process and the available functions are $\{f(x), g(x), h(x), i(x)\}$, then the function is replaced by either $g(x)$, $h(x)$ or $i(x)$, each with a 33% chance to be chosen.

2.1.2 Crossover. Since the architecture of the neural networks remains constant through the evolution process, crossover is rather straightforward. First, the proportion of individuals in the offspring produced via crossover is controlled by the crossover proportion parameter ($P_c \in [0.0, 1.0]$). For example, if $P_c = 0.3$, then 30% of the individuals in the offspring generation will be the result of crossover.

If an individual is chosen to be produced through crossover, then its links and hidden nodes are produced in the following way. Each link has a 50% chance to receive its weight from the first parent, and a 50% chance to receive its weight from the second parent. Likewise, each hidden node has a 50% chance of inheriting the first parent's activation function, and a 50% chance to inherit the second parent's activation function.

The resulting individual has, on average, 50% of the first parent's genotype and 50% of the second parent's genotype.

2.2 Experimental Setup

For the purposes of this manuscript, three classification datasets were tested. The datasets are all available through the Python `sklearn` library [6, 19]. The three sets used were the **`wine`**, **`breast cancer`**, and **`iris`** classification sets. Their main characteristics are presented in Table 1.

Each dataset was divided into three subsets: Training, Validation, and Testing. 60% of instances were used for training, 20% for validation, and the remaining 20% for testing. The results reported

Set #	Set Name	Instance Number (n)	Feature Number (m)	Class Number (c)
1	Iris	150	4	3
2	Breast Cancer	569	30	2
3	Wine	178	13	3

Table 1: The three datasets used in the experiments. All three datasets can be found in the `sklearn` library.

#	Name	Equation	Domain	Range
1	sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	$(-\infty, \infty)$	(0,1)
2	tanh	$f(x) = \frac{2}{1+e^{-x}} - 1$	$(-\infty, \infty)$	(-1, 1)
3	LReLU	$f(x) = \begin{cases} 0.01x & x \leq 0 \\ x & x > 0 \end{cases}$	$(-\infty, \infty)$	$(-\infty, \infty)$
4	ReLU	$f(x) = \begin{cases} 0.0 & x \leq 0 \\ x & x > 0 \end{cases}$	$(-\infty, \infty)$	$[0, \infty)$

Table 2: The functions used by the EvoNN algorithm

Parameter	Value
μ	50
λ	50
Weight Mutation Probability (P_m)	0.01
Weight Mutation Radius (R_m)	0.1
Function Mutation Probability (P_{m_f})	0.05
Crossover Proportion (P_c)	0.3
Elitism	1
Selection Method	Tournament (size=2)
Fitness Function	Logloss

Table 3: The evolutionary parameters for EvoNN

are based on the testing sets. The EvoNN framework was compared against an ANN framework which uses gradient descent and backpropagation [1].

For each dataset, 200 runs were performed with a different random seed each time, and the average of the 200 runs as well as the standard deviations were recorded and compared using the Wilcoxon Signed Rank test [28]. Within the 200 comparisons, the data was shuffled at random and split into training, validation, and testing subsets every 10 runs, creating 20 "mini-runs" of 10 repetitions each. This was done to simulate different distributions of classes within the subsets. Each algorithm was allowed to run until it showed no signs of improvement for 200 iterations[2] on the validation set, or until it reached 10,000 iterations.

2.2.1 Standard ANN Parameters. The standard ANN had a fixed architecture of three layers: m units for input (where m is the feature number for each set), 10 hidden units employing the sigmoid activation function, and c output units (where c is the number of classes for each set) employing the softmax activation function. A learning rate of 0.001 was used.

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{c} e^{x_j}} \tag{5}$$

The loss function chosen was the multiclass logloss function, defined as

$$logloss(Y, \hat{Y}) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{c} y_{ij} \times ln(\hat{y}_{ij}) \tag{6}$$

Where Y is the true value of the classes, and $\hat{Y}$ is the matrix of predictions generated by the neural network, y_{ij} is the true value of instance i belonging to class j (1 or 0 if it belongs, or doesn't belong, respectively), and $\hat{y}_{ij}$ is the estimated probability that instance i belongs to class j. To prevent computational failures, $\hat{y}_{ij}$ is taken to be $min(1-10^{-15}, max(10^{-15}, p_{ij}))$ where p_{ij} is the neural network's predicted probability that instance i belongs to class j.

2.2.2 EvoNN Parameters. EvoNN also had a fixed architecture of three layers: m input units, 10 hidden units, and c output units employing the softmax activation function. However, each of the 10 hidden units could have a different activation function. The activation functions provided as a function "bank" were the sigmoid function, the hyperbolic tangent (tanh) function, the Rectified Linear Unit function (ReLU) [9], and the Leaky Rectified Linear Unit (LReLU) function [15]. Their properties are outlined in Table 2. The evolutionary parameters are laid out in Table 3. For brevity purposes, the parameters are not described in this manuscript, but an excellent introduction to the hyper-parameters of evolutionary computation can be found in [11].

Notice that the logloss loss function which was used as the loss function for the standard neural network, is also used as the fitness function for EvoNN such that both methods are optimizing the same objective.

2.3 Custom Functions in EvoNN

When the main evolver (optimizer) is instantiated, users can provide custom functions for the evolver to use. The signature for EvoNN is as follows:

[1] The code for the standard ANN was taken from https://www.analyticsvidhya.com/blog/2017/05/neural-network-from-scratch-in-python-and-r/ and adjusted to be used with the logloss loss function

[2] 200 epochs for the neural network, and 200 generations for EvoNN

```
myEvo = Evolver( G=10000,
                 early_stopping=200,
                 MU=50,
                 LAMBDA=50,
                 P_m=0.01,
                 P_mf=0.01,
                 R_m=0.1,
                 P_c=0.3,
                 elitism=True,
                 tournament_size=2,
                 fitness_function=logloss,
                 final_activation_function=softmax,
                 additional_functions=[LReLU, ReLU])
```

The important two parameters to pay attention to are `additional_functions` and `fitness_function`.

The parameter `fitness_function` tells EvoNN what the objective function is to optimize. This function is used for both training and validation sets, and can be any arbitrary function with the following signature:

```
def myFunction(y_predicted, y_true):
      # myFitnes =  ...
      # Calculations of the loss between
      # predicted and known values go here


      return myFitness # the result of the fitness
                       # a single real value
```

Note that this allows non-differentiable measures such as the Area Under Curve (AUC).

The parameter `additional_functions` is in fact a list of activation functions which can be used within the hidden layer. These functions must all conform to the signature:

```
def myActivationFunction(x):
      # myOutput = ...
      # Calculations of the activation function
      # are performed here

      return myOutput
```

Under this signature, both `x` and `myOutput` are $n \times 1$ matrices where n is the number of samples in the set. In essence, this method applies the function to all examples in the set at once. When each unit applies its function, the result is an $n \times h_i$ matrix where n is the number of instances in the set, and h_i is the number of hidden units at layer i, i.e. the $n \times h_i$ matrix is the result of the sample set forward propagating through the i^{th} layer.

In the current set of experiments, the list of functions provided included the ReLU and LReLU functions. This is because by default, EvoNN has the sigmoid and tanh functions in its function collection.

2.4 Preprocessing, language, and hardware

All experiments reported have been performed on an Asus VivoBook with an i7-8550U processor. The code was written in Python 3.5.4.

All features were scaled into the range [-1.0, 1.0] by dividing the values of each feature i by $max(max(\vec{x}_i), |min(\vec{x}_i)|)$, where $\vec{x}_i$ is the vector of values in feature i.

3 RESULTS

3.1 Optimization Performance

Figure 5 and Table 4 present the resulting average values of logloss and the standard deviations associated with them. It is clearly evident that the EvoNN system performed superiorly to the standard ANN on average, with significantly smaller average logloss values and a p-value < 0.01 for all three datasets. However, it is also evident that the standard deviations for all three sets are rather large, more so for the standard ANN than for EvoNN, and these differences should be addressed.

In order to understand the observed results it is necessary to remember the data seen by the learning algorithms was presented as a set of different permutations with different distribution of the classes between the training, validation, and test sets. These different permutations create different fitness, or loss, landscapes for the learners to navigate through, creating more or less frequent local optima. Additionally, different permutations would change the extent the training set can be telling of the test set, because of different class distributions within the training and testing sets. This property could change the logloss values each of the learners can attain as it is training. Evidently, for most of the permutations, the EvoNN system outperforms the standard ANN.

These results are important since when constructing models, it is not always possible to attain an accurate representation of test sets due to limited amount of data, class imbalances, or other biases related to data gathering. Having confidence in the model's general flexibility towards distribution of examples and classes, in addition to confidence in its ability to stop before it begins overfitting is very important, and the results observed here are very encouraging regarding this aspect of EvoNN.

3.2 Run-time Performance

Figure 6 and Table 5 present the run time averages for both optimization systems. As expected, the run-time performance of the standard ANN is superior to the EvoNN optimizer. While the run time of both the standard ANN and EvoNN are dependent on their hyper-parameters, generally a decrease of run time will also compel a decrease of accuracy or performance. These results, combined with the results in Section 3.1 show that the superior performance observed with EvoNN has a trade-off with regards to run-time.

Generally, optimization problems can be divided into online and offline problems. Online problems have the data changing frequently (even so frequently that at every iteration, a new data point is added), and a learner must continuously adjust to maintain an optimal model of the data. Offline problems have the data changing infrequently, or even never, such that a model produced once can be used for an extended period of time. Online optimization problems require a fast run-time performance from the model, since it must adjust quickly to accurately reflect new information coming in. However, offline optimization instances can incur an additional run-time cost in exchange for a higher accuracy, or a less lossy, model.

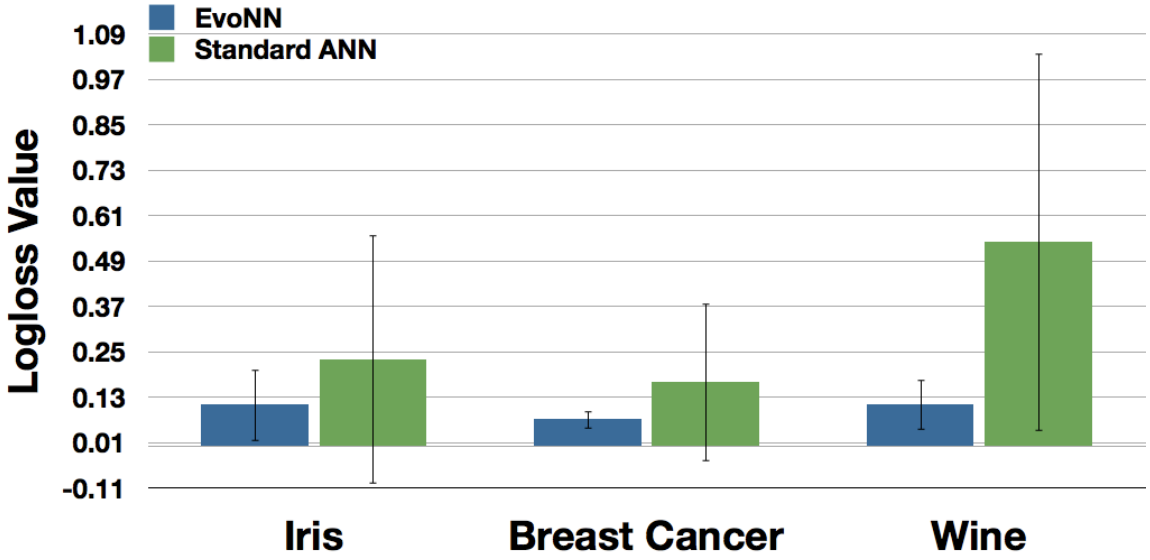


Figure 5: The logloss score of EvoNN versus the standard ANN on the test set for the three sample datasets. The averages of 200 runs, as well as the standard deviations, are shown.

Dataset	EvoNN		Standard ANN	
	Average Logloss	Standard Deviation	Average Logloss	Standard Deviation
Iris	0.11	0.096	0.23	0.33
Breast Cancer	0.071	0.025	0.17	0.21
Wine	0.11	0.067	0.54	0.50

Table 4: Optimization results of EvoNN and the standard ANN

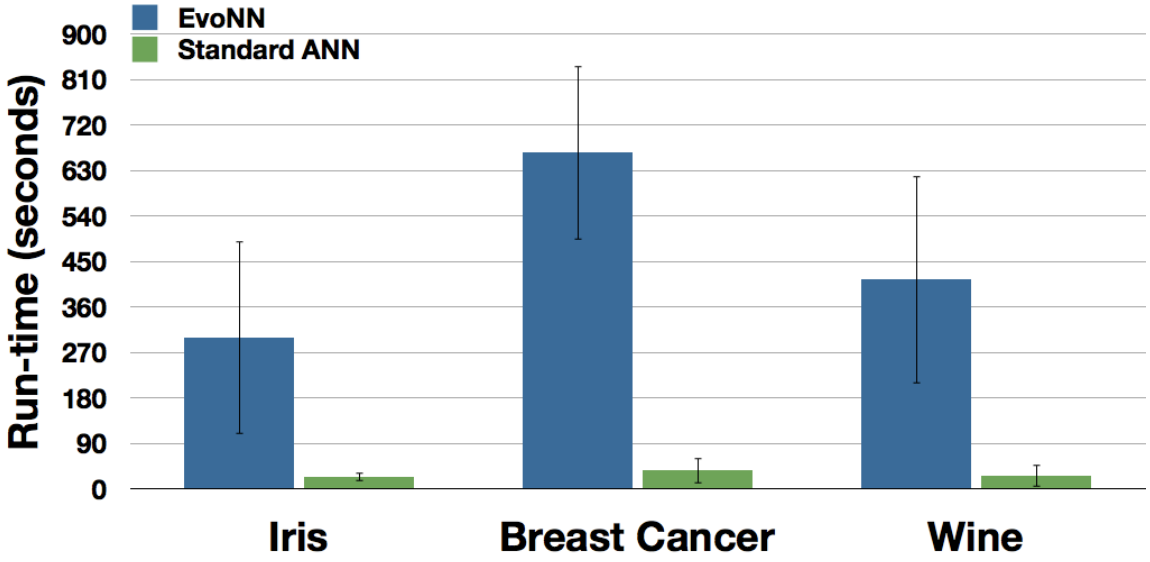


Figure 6: The run-time (in seconds) of EvoNN versus the standard ANN on the test set for the three sample datasets. The averages of 200 runs, as well as the standard deviations, are shown.

Dataset	EvoNN		Standard ANN	
	Average Run-time (seconds)	Standard Deviation	Average Run-time (seconds)	Standard Deviation
Iris	299.66	191.87	23.15	8.92
Breast Cancer	664.70	173.20	36.92	26.26
Wine	414.49	206.64	24.68	22.40

Table 5: Run-time results of EvoNN and the standard ANN

Furthermore, it is important to consider the architecture of both algorithms when observing run times. The nature of evolutionary algorithms is that many steps in the evolutionary process can be parallelized. Mutation, crossover, evaluation, and selection are all processes which work on the individual level and can be applied to several individuals in parallel. This feature is not available for the backpropagation algorithm. The backpropagation must happen sequentially since each iteration of the backpropagation depends on the new position of the model in the loss function space. This can, in theory, further decrease the run time and allow EvoNN to deliver superior results at a reduced run-time.

Generally, it may be the case that a fully optimized ANN which uses backpropagation will always deliver results faster than EvoNN. However, for applications where an increased accuracy or flexibility is important and can be traded for additional run-time, EvoNN shows superior results.

4 CONCLUSIONS AND FUTURE WORK

This article described a new system, EvoNN, which presents users with two novel features. First, in addition to the general ANN optimization method of adjusting the network's weights, the activation functions of the hidden layer are optimized as well. This presents an added degree of flexibility in the network's exploration of the fitness space presented to it within the problem. Second, EvoNN can accept as part of its arguments novel activation functions to be used within its "alphabet" of functions. The functions must conform to a particular signature, but are overall simple to implement, can have an arbitrary complexity or shape, and need not be differentiable.

Section 3 demonstrates the results of EvoNN when tested against a standard ANN on three classification sample sets. EvoNN shows superior performance with regards to optimization, but at the expense of increased run-time. This shows there exists a trade-off between obtaining higher accuracy results, and the run-time dedicated to obtain such accuracy.

However, it is important to remember there are several potential routes for accelerating the observed run times. First, as mentioned before, evolutionary algorithms can be parallelized easily for most instances since evaluation, crossover, mutation, and selection are all actions which operate on the individual level, thus they can be divided among the number of cores or threads available for the system. Additionally, since the algorithm is written using Python, it can in fact be accelerated by using Cython, a system which compiles Python code by the addition of static type decelerations (Python does not usually make static deceleration but rather resolves types during runtime).

Furthermore, EvoNN holds an additional advantage over standard ANN approaches. Since EvoNN does not use gradient directly, it does not require for the activation functions, or the fitness function, to be differentiable. This provides users with the ability to design arbitrary functions, or use optimization targets that are not necessarily differentiable.

Future work on EvoNN will look into a larger array of test sets to further validate the results observed in this manuscript. A combination of both regression and classification problems from different fields could be tested. In addition, the use of the Cython compiler for Python could be implemented to dramatically reduce the run-time of most EvoNN operations.

Further extensions to EvoNN could see the evolver adjusting not just the weights and activation functions of hidden units, but even the activation functions of the output layer (this is now a parameter for EvoNN). Furthermore, additional hyper-parameters could be incorporated as input to accommodate a potentially larger array of user specified functions (e.g. the Adaptive Piecewise Linear (APL) function described in [1]). Additionally, the evolver could also optimize the architecture of the network itself, deciding through evolution the appropriate size of each hidden layer, and the number of hidden layers. Such advances would have to resolve the issue of crossover between ANNs of different architecture, but if implemented could add another degree of flexibility and adaptability to EvoNN.

Overall, EvoNN shows promise in this work, and opens new research avenues into the optimization and automatic generation of ANNs.

REFERENCES

[1] Forest Agostinelli, Matthew D. Hoffman, Peter J. Sadowski, and Pierre Baldi. 2014. Learning Activation Functions to Improve Deep Neural Networks. *CoRR* abs/1412.6830 (2014). arXiv:1412.6830 http://arxiv.org/abs/1412.6830

[2] Fardin Ahmadizar, Khabat Soltanian, Fardin AkhlaghianTab, and Ioannis Tsoulos. 2015. Artificial neural network development by means of a novel combination of grammatical evolution and genetic algorithm. *Engineering Applications of Artificial Intelligence* 39 (2015), 1 – 13. https://doi.org/10.1016/j.engappai.2014.11.003

[3] Alberto Alvarez. 2002. A Neural Network with Evolutionary Neurons. *Neural Processing Letters* 16, 1 (01 Aug 2002), 43–52. https://doi.org/10.1023/A:1019747726343

[4] Marijke F. Augusteijn and Thomas P. Harrington. 2004. Evolving transfer functions for artificial neural networks. *Neural Computing & Applications* 13, 1 (01 Apr 2004), 38–46. https://doi.org/10.1007/s00521-003-0393-9

[5] P.G. Benardos and G.-C. Vosniakos. 2007. Optimizing feedforward artificial neural network architecture. *Engineering Applications of Artificial Intelligence* 20, 3 (2007), 365 – 382. https://doi.org/10.1016/j.engappai.2006.06.005

[6] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.

[7] Wlodzislaw Duch and Norbert Jankowski. 1999. Survey of Neural Transfer Functions. *Neural Computing Surveys* 2 (1999), 163–213.

[8] Dario Floreano, Peter Dürr, and Claudio Mattiussi. 2008. Neuroevolution: from architectures to learning. *Evolutionary Intelligence* 1, 1 (2008), 47–62.

[9] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep Sparse Rectifier Neural Networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research)*, Geoffrey Gordon, David Dunson, and Miroslav Dudík (Eds.), Vol. 15. PMLR, Fort Lauderdale, FL, USA, 315–323. http://proceedings.mlr.press/v15/glorot11a.html

[10] J J Hopfield. 1982. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences* 79, 8 (1982), 2554–2558. arXiv:http://www.pnas.org/content/79/8/2554.full.pdf http://www.pnas.org/content/79/8/2554

[11] Christian Jacob. 2001. *Illustrating evolutionary computation with Mathematica*. Morgan Kaufmann Pub.

[12] Donald E. Waagen John R. McDonnell. 1992. Evolving neural network architecture. (1992), 1766 - 1766 - 12 pages. https://doi.org/10.1117/12.130875

[13] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

[14] Y. Liu and X. Yao. 1996. Evolutionary design of artificial neural networks with different nodes. In *Proceedings of IEEE International Conference on Evolutionary Computation*. 670–675. https://doi.org/10.1109/ICEC.1996.542681

[15] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. 2013. Rectifier Nonlinearities Improve Neural Network Acoustic Models.

[16] G. Mani. 1990. Learning by gradient descent in function space. In *1990 IEEE International Conference on Systems, Man, and Cybernetics Conference Proceedings*. 242–247. https://doi.org/10.1109/ICSMC.1990.142101

[17] David J. Montana and Lawrence Davis. 1989. Training Feedforward Neural Networks Using Genetic Algorithms. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1 (IJCAI'89)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 762–767. http://dl.acm.org/citation.cfm?id=1623755.1623876

[18] Varun Kumar Ojha, Ajith Abraham, and Václav Snášel. 2017. Metaheuristic design of feedforward neural networks: A review of two decades of research. *Engineering Applications of Artificial Intelligence* 60 (2017), 97 – 116. https://doi.org/10.1016/j.engappai.2017.01.013

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[20] A. Sedki, D. Ouazar, and E. El Mazoudi. 2009. Evolving neural network using real coded genetic algorithm for daily rainfall-runoff forecasting. *Expert Systems with Applications* 36, 3, Part 1 (2009), 4523 – 4527. https://doi.org/10.1016/j.eswa.2008.05.024

[21] Shinichi Shirakawa, Yasushi Iwata, and Youhei Akimoto. 2018. Dynamic Optimization of Neural Network Structures Using Probabilistic Modeling. *CoRR* abs/1801.07650 (2018). arXiv:1801.07650 http://arxiv.org/abs/1801.07650

[22] Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. 2009. A Hypercube-based Encoding for Evolving Large-scale Neural Networks. *Artif. Life* 15, 2 (April 2009), 185–212. https://doi.org/10.1162/artl.2009.15.2.15202

[23] Kenneth O. Stanley and Risto Miikkulainen. 2002. Efficient Reinforcement Learning Through Evolving Neural Network Topologies. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*. Morgan Kaufmann, San Francisco, 9. http://nn.cs.utexas.edu/?stanley:gecco02b

[24] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation* 10, 2 (2002), 99–127. https://doi.org/10.1162/106365602320169811 arXiv:https://doi.org/10.1162/106365602320169811

[25] Andrew James Turner and Julian Francis Miller. 2014. NeuroEvolution: Evolving Heterogeneous Artificial Neural Networks. *Evolutionary Intelligence* 7, 3 (01 Nov 2014), 135–154. https://doi.org/10.1007/s12065-014-0115-5

[26] D. V. Vargas and J. Murata. 2017. Spectrum-Diverse Neuroevolution With Unified Neural Models. *IEEE Transactions on Neural Networks and Learning Systems* 28, 8 (Aug 2017), 1759–1773. https://doi.org/10.1109/TNNLS.2016.2551748

[27] Lin Wang, Yi Zeng, and Tao Chen. 2015. Back propagation neural network with adaptive differential evolution algorithm for time series forecasting. *Expert Systems with Applications* 42, 2 (2015), 855 – 863. https://doi.org/10.1016/j.eswa.2014.08.018

[28] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. http://www.jstor.org/stable/3001968