Journal of Machine Learning Research 21 (2019) 1-10 Submitted 11/19; Published 12/19

Comparison of Various Learning Algorithms for Training Multilayer Perceptron Neural Networks

Brandon Sladek BRANDONSLADEK@GMAIL.COM
School of Computing
Montana State University
Bozeman, MT 59718, USA

Jared Thompson J.A.THOMPSON22@GMAIL.COM
School of Computing
Montana State University
Bozeman, MT 59718, USA

Kyle Hagerman HAGERMANKYLE96@GMAIL.COM
School of Computing
Montana State University
Bozeman, MT 59718, USA

Ryan Hansen RYANHANSEN2222@GMAIL.COM
School of Computing
Montana State University
Bozeman, MT 59718, USA

Abstract

In this research we compare the performance of four different learning algorithms for training a multilayer perceptron neural network: backpropagation, a genetic algorithm, differential evolution, and particle swarm optimization. We implemented the learning algorithms in Python, and compared their performances on classification and regression data sets from the UCI machine learning repository (UCI (2019)). Overall, given the data sets used, we found that the standard backpropagation algorithm gave the best results both in terms of accuracy/error and convergence time.

Keywords: Neural Network, Gradient Descent, Backpropagation, Genetic Algorithms, Differential Evolution, Particle Swarm Optimization, Parameter Learning Algorithms

1. Introduction

Artificial neural networks have quickly become one of the most popular machine learning strategies in industry today. Their ability to learn complex decision boundaries for a wide variety of problems has led to the successful application of neural networks in fields as diverse as medical imaging and financial forecasting. This research will compare four popular training procedures for Neural Networks: (1) Backpropagation, (2) Genetic Algorithm, (3) Differential Evolution, and (4) Particle Swarm Optimization.

2. Problem

In ”Multilayer Feedforward Networks are Universal Approximators”, it is proposed that, in theory, a neural network can be built and trained to any degree of accuracy, provided there is an actual relationship between the input and expected output, and the network is built in such a way that it can adequately learn the features of the input space and map them to an accurate output space with respect to the expected output. The authors claim that “failures” in the networks can be attributed to poor training methods, poor choices for hyperparameter values, or simply a lack of a deterministic relationship between the input and the expected output.

If we accept the claims made about multilayer feedfoward networks, its success is contingent upon how the neural network gets trained - the question becomes, what are the best methods? Or, taken from a different point of view, is there even one “best” training solution, or is it some combination of each or some of them? By comparing all four training methods with varying hyperparameters we aim to develop a better understanding of this problem, and expand upon in the final sections of this paper.

3. Multilayered Perceptron (MLP)

This is a special case of a feedforward neural network. In this network, we get to choose an arbitrary number of layers, the number of nodes in each layer, and the activation functions. The entire network is trained with gradient descent using backpropagation.

Overview The MLP is perhaps one of the most general implementations of a feedforward network. The idea is to stack an arbitrary number of layers of nodes to try to capture patterns in data.

Formalism The main theory behind the MLP is encompassed in its activation functions. Common activation functions intend to normalize the activation of each node between 0 and 1, examples being the Sigmoid and ReLU functions. This means a low activation should have negligible impacts on the output, and high activation should have larger effects.

Our Implementation We implemented the option to use both ReLU or the Sigmoid, but ran experiments with Sigmoid. Then, we varied the hyperparameters - number of nodes in each layer and number of layers - in order to find good results with respect to our loss functions.

4. MLP Training Algorithms

4.1 Representation

Each of the following algorithms are founded on the idea that we can represent the object we want to optimize as a point, x, in some (often times) high dimensional euclidean space $\mathbb{R}^d$. If we can identify some function $f : \mathbb{R}^d \to \mathbb{R}$ that indicates the fitness of each point, we can then try to search the space for the optimum object. In this experiment, our objects are neural networks that live in $\mathbb{R}^d$, where $d = |weights| + |biases|$. This means each weight and bias have their own corresponding dimension in this representation. Finally, our fitness function, f, is closely related to the accuracy/cost of each neural network.

4.2 Fitness

We define the fitness of a given neural network $x \in \mathbb{R}^d$ in two ways, depending on if the data set is a regression set or a classification set.

4.2.1 Regression

Recall a neural network can be thought of as a function $N : Input \rightarrow Class$ from some input vector to its class value. In regression data, that class value is usually some real number. Ideally, a perfectly trained neural network will have 0 error on each output value. Thus, if i is some input vector with corresponding class $class(i)$, we want to minimize $N(i) - class(i)$ for each data point. The easiest way to do this is through minimizing the sum of squared errors, $\Sigma(|N(i) - class(i)|)^2$, to avoid issues with negatives. Lastly, since we chose to make maximization algorithms, we translate the problem to a maximization problem by taking the inverse. This brings us to our final result for regression fitness: For some neural network x, and a set of input data points T, the fitness of x,

$$f(x) = \frac{1}{\Sigma_{y \in T}(|N(y) - class(y)|)^2} \tag{1}$$

4.2.2 Classification

We originally calculated fitness for classification data the same as regression, however we ran into an issue where a low error did not necessarily correspond to the highest accuracy of the neural network. We remedied this issue by defining fitness as follows. Recall the 0-1 loss function

$$L(y) = \left\{ \begin{array}{cc} 1 & N(y) = class(y) \\ 0 & otherwise \end{array} \right. \tag{2}$$

$$f(x) = \sum_{y \in T} L(y)/|y| \tag{3}$$

This is just the accuracy of N on the given data set.

4.3 Backpropagation

The network is initialized with arbitrary weights and biases, along with the chosen activation functions for each layer. The network is trained through supervised learning. Given a training data point x, with desired output value Y, there will be a subsequent output activation in the output nodes $N(x)$. To train the network, we take the squared error, $Err = (Y - N(x))^2$, and attempt to minimize it through a gradient learning algorithm - backpropagation. To do this, we optimize ∇Err, where we are taking derivatives with respect to weights and biases. The goal is to learn the weights and biases that minimize the squared error. Subsequently, we update each weight and bias by the following formula (substitute w with b if updating bias).

$$\mathbb{W}_{new} = \mathbb{W}_{old} - \alpha \nabla Err \tag{4}$$

Where α is a tuned learning rate. We propagate through each layer, until each weight and bias in our network has been updated. Finally, it is proven that given enough nodes and layers, any function can be approximated to infinite precision using one of these neural nets (J.R.M.Smiths) (Hornik et al. (1989)).

4.4 Genetic Algorithm (GA)

The Genetic Algorithm was implemented as its own class, and works in conjunction with the original NeuralNetwork and ExperimentRunner classes. All GA specific functionality is delegated across four methods: create_roulette_wheel(), get_parents(), do_variations(), and do_GA(). The algorithm is executed in ExperimentRunner, which calls an instance of NeuralNetwork that creates a population of NeuralNetwork instances given a population size parameter and a hidden layer specification. That initial population is passed into the do_GA() method, where the rest of the methods get executed.

GA “learns” through a population of individual neural networks over multiple generations. Each individual in the network makes a guess, and the accuracy of the guesses across the training data corresponds to the individual’s fitness score and ability to reproduce. The do_GA() method logic is implemented in this sequence - (1) find the fitness of every member of the current population. (2) create a roulette wheel corresponding to the current population, then (3) create a new population of children, identical to the size of the old population, where each child’s parents are drawn from two random casts on the roulette wheel, and (4) set the child population to the current population and repeat until we have reached the maximum number of generations. In GA, there is only two sources of change - mutation and crossover. In our approach, we implemented total generational turnover, so the subsequent population was comprised solely of children of parents from the previous generation. The crossover rate was set exactly at .5, and the mutation rate was varied between .001 and .005. For crossover to work, it implies that fit parents produce more offspring than non-fit parents - the logic of which is implemented in the roulette wheel. For population of varying size, take for example 50, our roulette wheel would have 50 ”slots” for the most fit individual, 49 the second most, and so on and so forth.

4.5 Differential Evolution (DE)

The third algorithm implemented in this experiment is Differential Evolution. It is a Parameter Learning algorithm inspired by biological methods, similar to the Genetic Algorithm but with key differences (Abdul-Kader (2009)).

Differential Evolution (DE) operates on a population of neural networks (Ilonen et al. (2003)). Each network is an individual, and treated as if it were a biological organism. These individuals are used to breed and create offspring networks intended to increase the fitness of the population over time. Fitness is the measure of accuracy a given neural network has. The population will iterate through a number of generations with the following steps performed on each individual: Selection/Mutation, Crossover, and Replacement (Ilonen et al. (2003)).

The GA implements a point mutation method but DE combines mutation directly into selection. A roulette wheel like the one described in 4.2 is used to randomly pick 3 individuals from the population then a mutation vector (Slowik and Bialko (2008)) is created, see the following equation:

$$v = r_1 + F(r_2 - r_3)$$

Where v is the mutation vector, r_1, r_2, r_3, are random neural nets chosen from the roulette wheel array, and F is a scaling factor (a random number between 0 and 2).

Once the mutation vector has been created, uniform crossover occurs between it and the current individual, the parent, at a chance of 50% of the weights or biases switching places. The resultant vector is the child. Replacement will decide whether the parent or the child will remain in the population. Whichever has a greater fitness will remain to increase the average population fitness (Abdul-Kader (2009)). After a set number of iterations, the individual with the highest fitness is chosen to classify the test set and a final accuracy for the process is received.

The DE algorithm is implemented with its own class that runs each step in a separate method. A main method run_de() handles the while loop that counts iterations and displays the status of the population after each generation. This method calls all other methods in the class. The roulette_wheel() method generates a weighted array based on highest fitness in the population the same as in 4.2. The de_selection() method finds the mutation vector from the equation above and performs crossover. For crossover, a random number is generated between 0 and 1. If it is higher than .5, then the values of the given index are swapped. Else, no crossover occurs. Then we find which individual remains in the population with the de_replacement() method. The fitness of the child vector is calculated and if it is better than its parent, the parent will die and the child will take its place. Otherwise, the parent lives on. In this way, the best "genetic material" will be passed on.

4.6 Particle Swarm Optimization (PSO)

This is an optimization technique that is based on swarm thinking. The idea is for a population to find optima and use both personal and communal data to avoid common pitfalls of standard hill climbing methods (i.e. getting stuck in local optima).

4.6.1 Swarm

A swarm is basically a data structure for particles. It handles most of the updating for running the program, and contains one significant parameter - Global Best Location, denoted $GBL()$. This is a quantity that helps each particle make educated decisions on how to search through the optimization space. Intuitively, this is the most fit neural network that the algorithm has found so far while executing.

4.6.2 Particles

A particle is an abstract idea meant to represent the object we are optimizing. It has four main pieces of data.

Position: The position of a particle represents the current object, denoted $pos()$. In this experiment, the position is a neural network with set values for its weights and biases. We represent the position as a point in some high dimensional euclidean space as described in section 4.1. If p is a particle representing $x \in \mathbb{R}^d$, $pos(p) = x$

Best Known Location A particle basically just moves around the search space with some educated guessing for where to search. To help generate this educated guessing, we store the particle's best known location, denoted $BKL()$. If p is a particle, let $M(p)$ be the set of all $pos(p)$ it has been to thus far. Then $BKL(p) = argmax(f(pos(p)))$. This is the neural network with the highest fitness.

Velocity The velocity of a particle is how it searches the optimization space. It has three major components that try to influence the particle to go into places likely to have high fitness.

1. *Inertia*: Inertia is exactly as it sounds. It is a continuation of prior velocity. It is governed by the tuning parameter $w \in [.4, .9]$ and its contribution to the final velocity is as follows. Let V_{Prev} denote the previous velocity of a particle. Then

$$V_{Inertia} = wV_{Prev} \tag{5}$$

2. *Personal*: This component of velocity describes a particle's bias to search around where it personally found its best results. This is governed by two parameters, one a random variable, $a \in (0, 1)$ and second a bias coefficient, a tuning parameter $C1$ that determines how influential you want the personal component to be. For a particle p, we define its personal velocity

$$V_{Personal} = a * C1 * (BKN(p) - pos(p)) \tag{6}$$

3. *Social*: This component describes a particle's bias to search around where it globally has heard of the best fitness. This is governed by two parameters, one a random variable, $b \in (0, 1)$ and second a bias coefficient, a tuning parameter $C2$ that determines how influential you want the social component to be. For a particle p, we define its social velocity

$$V_{Social} = b * C2 * (GBL(p) - pos(p)) \tag{7}$$

4. *Total*: The total velocity

$$V_{Total} = V_{Social} + V_{Personal} + V_{Inertia} \tag{8}$$

5. Notes: For convergence, $C1 + C2 \leq 4$, and we additionally set a maximum velocity. Otherwise, it is common for velocities to run off as the algorithm goes on.

The algorithm runs by initializing a set of particles randomly throughout the search space and has them explore via the velocity update strategy described above. We chose to have the algorithm end after a pre-determined number of iterations, as our personal computers did not have the power/time needed to run the algorithm until convergence.

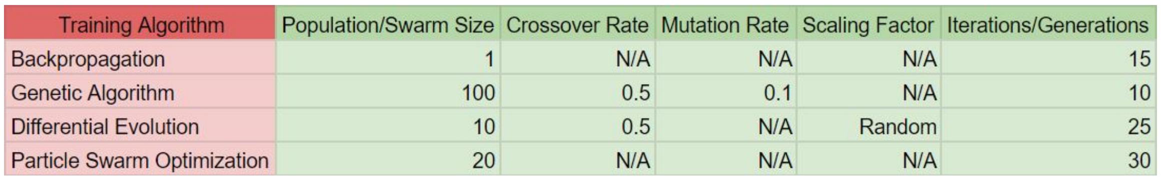
5. Experiment

In this section we explain our general approach for accepting or rejecting our hypothesis for the various data sets, network configurations, and corresponding parameter values.

5.1 Approach

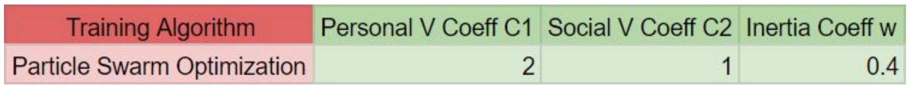
To run a specific experiment, we configure the ExperimentRunner class to use a specific algorithm, network configuration, data set, and various other hyperparameters and learning parameters (e.g. population size) when necessary. The ExperimentRunner class interfaces with each of our other classes to preprocess the data so it can be ran through each neural net, generates cross-validation folds to verify statistical significance of our results, and runs the specified algorithm on the data set. The output from each algorithm is a neural network corresponding to the network the algorithm converged to, hopefully the most fit network.

We run the test data (from the CV partitions) on the final neural network to yield a final result comparable through loss functions. For classification data, this score is an accuracy, and for regression it is a mean squared error value. Additionally we record the time it took for each algorithm to finish. We tuned the parameters by testing the results on 1 CV fold, because these algorithms take time to run. Additionally, we restricted each algorithm to run on one network configuration. Again, this is because of running time constraints. Finally, we compare the results of each algorithm.



Training Algorithm	Population/Swarm Size	Crossover Rate	Mutation Rate	Scaling Factor	Iterations/Generations
Backpropagation	1	N/A	N/A	N/A	15
Genetic Algorithm	100	0.5	0.1	N/A	10
Differential Evolution	10	0.5	N/A	Random	25
Particle Swarm Optimization	20	N/A	N/A	N/A	30

Figure 1: Table of parameter settings by algorithm.



Training Algorithm	Personal V Coeff C1	Social V Coeff C2	Inertia Coeff w
Particle Swarm Optimization	2	1	0.4

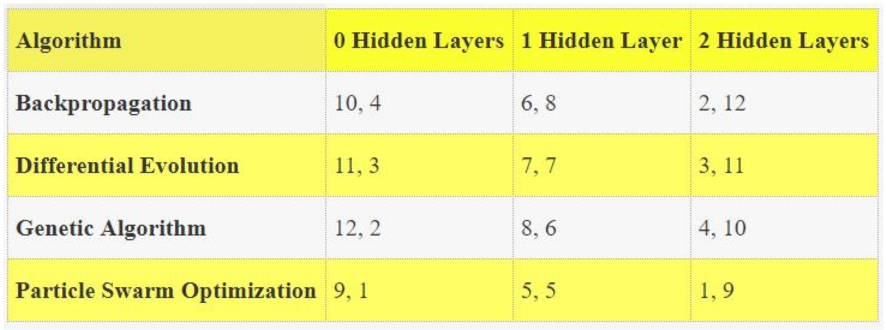
Figure 2: Table of PSO specific parameter settings.

5.2 Hypothesis

We predicted that with less hidden layers each algorithm will perform faster but with less accuracy. We also predicted that ParticleSwarmOptimization will be the most accurate with 2 hidden layers. Backpropagation will be the slowest training algorithm and ParticleSwarmOptimization will be the fastest. We predicted the GeneticAlgorithm will be just slightly faster than DifferentialEvolution but less accurate because it draws from less genetic material. See Figure 3 for the summary of our accuracy and performance hypotheses.

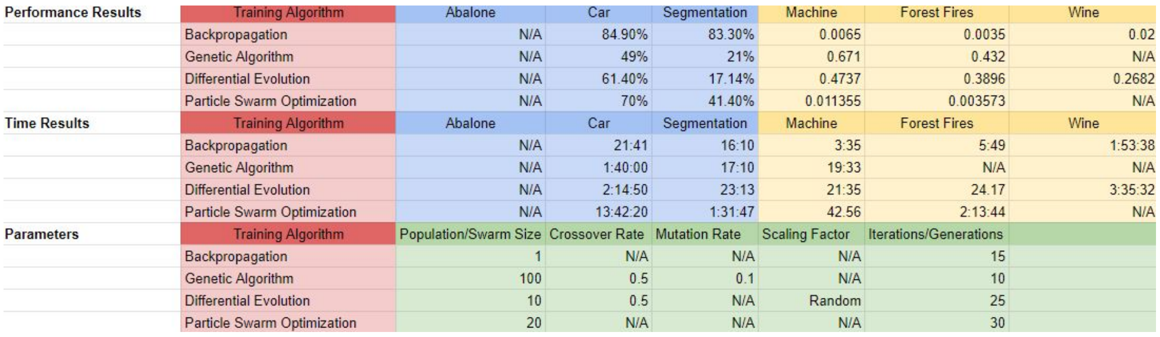
5.3 Results

See results in Figure 4 below.



Algorithm	0 Hidden Layers	1 Hidden Layer	2 Hidden Layers
Backpropagation	10, 4	6, 8	2, 12
Differential Evolution	11, 3	7, 7	3, 11
Genetic Algorithm	12, 2	8, 6	4, 10
Particle Swarm Optimization	9, 1	5, 5	1, 9

Figure 3: Table depicting our performance hypotheses. The first number of each cell is the ranking of the algorithm and hidden layers on our predicted accuracy with 1 being the most accurate and 12 being the least accurate. The second number in each cell is our prediction for running time, again with 1 being the fastest and 12 being the slowest.



Performance Results	Training Algorithm	Abalone	Car	Segmentation	Machine	Forest Fires	Wine
	Backpropagation	N/A	84.90%	83.30%	0.0065	0.0035	0.02
	Genetic Algorithm	N/A	49%	21%	0.671	0.432	N/A
	Differential Evolution	N/A	61.40%	17.14%	0.4737	0.3896	0.2682
	Particle Swarm Optimization	N/A	70%	41.40%	0.011355	0.003573	N/A
Time Results	Training Algorithm	Abalone	Car	Segmentation	Machine	Forest Fires	Wine
	Backpropagation	N/A	21:41	16:10	3:35	5:49	1:53:38
	Genetic Algorithm	N/A	1:40:00	17:10	19:33	N/A	N/A
	Differential Evolution	N/A	2:14:50	23:13	21:35	24.17	3:35:32
	Particle Swarm Optimization	N/A	13:42:20	1:31:47	42.56	2:13:44	N/A
Parameters	Training Algorithm	Population/Swarm Size	Crossover Rate	Mutation Rate	Scaling Factor	Iterations/Generations	
	Backpropagation	1	N/A	N/A	N/A	15	
	Genetic Algorithm	100	0.5	0.1	N/A	10	
	Differential Evolution	10	0.5	N/A	Random	25	
	Particle Swarm Optimization	20	N/A	N/A	N/A	30	

Figure 4: Table of all training algorithm results. Blue columns are classification data sets measured by accuracy. Yellow columns are regression data sets measured by squared error. Green columns denote tuning parameters, and red columns specify the algorithm.

6. Analysis

General Analysis: We note that our final results did not compare with the expected results of these algorithms. We think this may be because the search space was too large for our initial randomized vector. As each of the algorithms (aside from backpropagation) use some form of genetic combination to find new neural networks, having little genetic diversity at the beginning may hamper our results significantly. Additionally, we think a lack of computation power may influence our results. We only tested on up to 30 iterations of each program, and while we did see steady increase throughout, the algorithms terminated before anything good could be reached.

Genetic Algorithm: **GA Performance** - GA had an average accuracy of about .35, and an average means squared error .551, meaning that across the board it was the 3rd or 4th worst performing algorithm. This was expected and consistent with the hypothesis made prior to running the experiments. However, the results are quite low for some of the data samples, and an explanation is warranted. For GA, each individual neural network within a population can change only by mutation and crossover. For mutation, this kind of change happens infrequently, and is not strategically selected; it is simply random. The only possibility then for mutation to have a positive impact on the model is from multiple generations across a large population size. In natural selection unfavorable mutations tend to lead to offspring death, thereby ensuring those genes will not get passed on. This did not occur in our experiment because of the way the roulette wheel was implemented, so there was always the possibility (smaller though it was) that an unfavorable mutation could be passed to the next generation. We found an optimum to be about .001.

The primary medium through which GA carries out "learning" is crossover. In our implementation, this is carried out by the roulette wheel, which ensures that fitter parents produce more offspring. The theory is that if we recombine each fit parents' chromosome of weights and biases, across multiple generations, we will eventually converge to an optima. There are two fundamental things to note here: First, this theory is predicated on the assumption that the initial population of parents represents the entire search space; if it does not, than the algorithm will converge on a local max/min (unless an extremely unlikely series of mutations occur). For our algorithm, the initial population was weighted and biased randomly, and this bounds our solution to the best neural network within the initial population. Furthermore, our implementation of the roulette wheel (see above) was, sub-optimal because it only slightly biases the network to retrieve more fit parents. According to (Siddique and Tokhi (2001)), GA is best used in conjunction with other algorithms as a way to search across models that have already been trained, thereby offering GA a better characterization of the search space, and already trained models, from which to perform crossover on. This is consistent with the results we observed and the above analysis.

GA Convergence Time - GA convergence time was one of the fastest, which makes sense given that GA had the least operations to perform.

Differential Evolution: **DE Performance** - Results for DE are mixed at best. With over 60% accuracy for the car dataset but lesser for every other dataset is far from ideal. Regression datasets all returned errors lesser than .5 with the Wine dataset being the best at an error of .2682. In previous projects, segmentation has been known to classify very well so the 17.14% is quite unexpected.

Differential Evolution's performance clearly hinges on number of iterations. Though above reported data only shows generations of 25, other tests were conducted to determine how to best achieve results. Upon increasing the number of generations, the fitness steadily increases with each iteration. This propels the population towards better accuracy or lesser error. A standard generation was chosen because in the case of the abalone dataset, the runtime increases drastically with any greater number of generations. Future studies in code optimization and greater number of generations with smaller datasets would hypothetically illustrate higher accuracies and lower error. **DE Convergence Time** - Running these parameters on datasets segmentation, machine, and forestfires took approximately 20-30 minutes. Datasets car, wine, and abalone took approximately 2-4 hours. There are several opportunities for code optimization that could reduce these runtimes allowing greater accuracies to be found in less time. For example, if the code were written to use Python multiprocessing to utilize multiple cores, then runtime would be cut down by a fraction of the number of cores used. Each generation could be run that much faster and we could run more iterations to find a better solution.

Particle Swarm Optimization: **PSO Performance** - Particle swarm tended to yield moderate accuracy for classification data sets, and competitive mean squared error for regression data sets. This puts it in a close second behind backprop. We would expect this to improve if we had more computation power and could run more particles and iterations in each run. Running on 30 iterations did steadily increase performance, but it did not have enough time to hit really good result values. **PSO Convergence Time** - Particle swarm took a long time to run. However, the long run times did seem to have an impact on resulting performance. In theory, we are performing just as many position updates on each of the new algorithms, so the time discrepancy is most likely computer related.

7. Conclusion

Neural networks offer an extremely flexible approach to modeling labeled data and predicting unlabeled data. Beyond the flexibility of the networks themselves, the training approach can be implemented in many different ways, some better than others for any particular problem. In our research, given the data sets used, we found that the standard backpropagation algorithm gave the best results both in terms of accuracy/error and convergence time.

References

HM Abdul-Kader. Neural networks training based on differential evolution al-gorithm compared with other architectures for weather forecasting34. *International Journal of Computer Science and Network Security*, 9(3):92–99, 2009.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Jarmo Ilonen, Joni-Kristian Kamarainen, and Jouni Lampinen. Differential evolution training algorithm for feed-forward neural networks. *Neural Processing Letters*, 17(1):93–105, 2003.

L.M.C. Buydens G.Kateman J.R.M.Smiths, W.J.Melssen. Using artificial neural networks for solving chemical problems: Part i. multi-layer feed-forward networks. *Using artificial neural networks for solving chemical problems: Part I. Multi-layer feed-forward networks*, 22:165–189.

MNH Siddique and MO Tokhi. Training neural networks: backpropagation vs. genetic algorithms. In *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No. 01CH37222)*, volume 4, pages 2673–2678. IEEE, 2001.

Adam Slowik and Michal Bialko. Training of artificial neural networks using differential evolution algorithm. In *2008 conference on human system interactions*, pages 60–65. IEEE, 2008.

UCI. UCI machine learning repository, 2019. URL `https://archive.ics.uci.edu/ml/datasets/`.