# An Evolutionary Training Algorithm for Artificial Neural Networks with Dynamic Offspring Spread and Implicit Gradient Information

Martin Ruppert
Institute of Computer Science
Worms University of Applied Sciences
Worms, Germany
ruppert@fh-worms.de

Eric MSP Veith, Bernd Steinbach
Institute of Computer Science
Freiberg University of Mining and Technology
Freiberg, Germany
{veith, steinb}@informatik.tu-freiberg.de

***Abstract*—Evolutionary training methods for Artificial Neural Networks can escape local minima. Thus, they are useful to train recurrent neural networks for short-term weather forecasting. However, these algorithms are not guaranteed to converge fast or even converge at all due to their stochastic nature. In this paper, we present an algorithm that uses implicit gradient information and is able to train existing individuals in order to create a dynamic reproduction probability density. It allows us to train and re-train an Artificial Neural Network supervised to forecast weather conditions.**

***Keywords*–artificial neural network; evolutionary algorithm; weather forecasting; smart grid.**

## I. Introduction

Machine learning finds its application in many areas. One of them is short-term weather forecasting, which is useful for predicting the output of renewable energy sources [1]. The basic assumption that wind speed or solar radiation follow a particular, detectable pattern introduces Artificial Neural Networks (ANN) as a probable device for forecasting. The simplest form of the ANN, the Perceptron, is primarily usable for detecting static patterns.

However, the more variety input data has, the larger the error of a Perceptron. Introducing ANNs with a short-term memory that implement the concept of time, such as Elman's [2], increase the success of the ANN.

Traditional training methods based on backpropagation, such as RPROP [3] and its variants, allow fast weight updates in online or stochastic mode, i.e., immediately after a pattern has been seen. However, these algorithms can get stuck in local minima. Evolutionary algorithms [4] can solve this problem by introducing randomness through their process of mutation and crossover, which also includes seemingly bad individuals. This offers a chance to escape a local minimum. However, this randomness typically increases the time it takes for an evolutionary algorithm to arrive at a properly trained ANN, and it can even be unsuccessful.

In this paper, we present a training method that combines the gradient descent technique of a backpropagation-based training method with the resilience of an evolutionary algorithm against local minima.

## II. Motivation

Since the search space during the training for artificial neural networks is big for any real-life application, many training functions harbor the danger of getting stuck in local minima. Evolutionary training methods circumvent this by introducing randomness into the process. However, this, in turn, increases the training time and does not guarantee success per se. Results obtained by training using evolutionary algorithms can even yield worse results, since, by purpose, there is no knowledge of an error gradient included.

This problem becomes apparent when using artificial neural networks for weather forecasting since the search space of a wind profile offers many local minima.

In [5], Maqsood *et al.* use an ensemble of neural networks to forecast weather. Although this ensemble technique is successful, it requires four different networks in order to yield these results. Moreover, the networks are retrained for the four seasons (winter, spring, summer and fall) separately. For this supervised training, they use a hand-selected sample set.

If forecasters are installed at different sites, supervised training will have to occur separately for each node, because their different locations mean different weather conditions. The ANN will have to continuously adapt itself in order to remain reliable even under uncommon weather conditions. Since remote sites such as wind parks will typically feature embedded systems, training should occur in a short period of time. However, using backpropagation-based algorithms will yield non-optimal results due to the algorithm getting stuck in local minima, which will introduce large deviation in cases of, e.g., gusts of wind.

To address this problem, we propose a combination of both evolutionary training and deterministic training algorithms that can use the advantages of both approaches. Our approach, which employs evolutionary strategies, uses information about the current success and implicit gradient information when creating the offspring. Furthermore, we allow existing individuals to be improved instead of resorting to improve the overall population through the offspring only.

The remainder of this paper is structured as follows. We describe the training algorithm in the Section III along with a pseudo-code representation in Figure 2. We discuss our approach in Section IV and conclude in Section V.

## III. The Training Algorithm

The algorithm currently finds its application in training neural networks that follow the design of Elman [2]. The difference to Elman's design lie in the connections to the context layer: The hidden layer is fully connected to the

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-357-5 

context layer. Furthermore, these connections can be trained, i.e., their weight is not set fixed to 1.0.

To the algorithm, the concepts of "neural networks" with "trainable weights" do not exist. Instead, we operate on individuals that we call *objects*. These objects, in turn, that have *parameters*. This application-agnostic approach is consistent with literature. It will, in the future, also allow us to apply the algorithm to other problems instead of constraining it to artificial neural networks. For ANNs, a parameter is a particular, trainable weight.

Additionally to their parameters vector, each object also has a *scatter* vector, **s**. It vector limits the interval of modification during an iteration, $t$, for each parameter $p_i$:

$$p_{t,i} = [-s_i \cdot p_{t-1,i}, s_i \cdot p_{t-1,i}] \quad (1)$$

As soon as an object is evaluated, its fitness is stored in its *fitness* vector. The total mean error is stored in $f_0$, while the mean error values of different samples are stored in $f_1, \ldots, f_n$.

Each object has additionally a maximum *age* that limits the number of iterations it may exist.

Since many steps of the training process require random numbers, we define a function called *frandom()* that returns a random number in the interval $[0.0, 1.0)$ with an uniform distribution. We can create different points of high density of the uniform distribution by calling *frandom()* multiple times. The calls are concatenated by addition or subtraction, depending on where we want these points to be. This uniform distribution is used to pick initial values for scatter and parameters for all objects derived from the user-supplied base object during the creation of the initial population.

We begin by creating the starting population. Each population consists of a number of objects that are active, i.e., trainable. This upper bound of the size of a population is contained in the variable *numActiveObjects*.

The training algorithm has the notion of an elite, i.e., a number of objects that are considered to be better than the rest of the population. The elite is included in the maximum number of objects in the population, i.e., $population = elite \cap others$. The size of the elite never changes: This is a user-configurable value. However, as soon as any other object outside the elite is better than any elite object, it is exchanged with the worst elite object.

The initial set of parameters is supplied by the user. The initial scatter vector is filled with random numbers within bounds supplied by the user. However, only the first, i.e., the *base object* $o_0$, of the initial population gets these pristine values assigned; for all other generated objects ($o_n$) these are modified according to Equations 2 and 3.

$$o_n.s_i = o_0.s_i \cdot \exp(4.0 \cdot (0.5 - frandom())) \quad (2)$$

$$o_n.p_i = o_0.p_i \cdot s_i - \sum_{0}^{3} frandom() \quad (3)$$

The user also supplies the fitness function, *fitness(o)*. It is required to return the fitness value of the object, $o$, as a single float-point number in order to compare it to the user-supplied target fitness. The fitness value is also used to sort the population. Since our training strategy is application-agnostic, it does not evaluate the ANN directly; this is the user's responsibility.

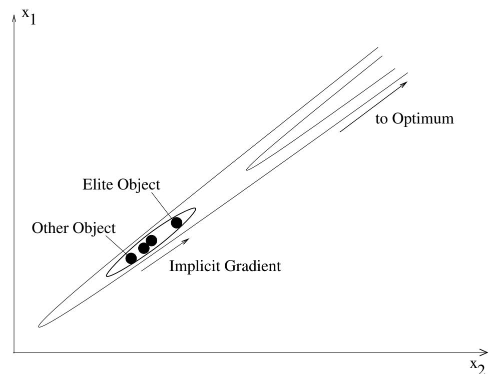


Figure 1. The implicit gradient information used by the modified reproduction function, *GenerateObject()*

To finalize the initialization phase, the population's fitness values are determined and the population is sorted accordingly.

The population is then iteratively improved until either the maximum number of iterations is reached (*maxIter*), there has been no improvement for a designated number of iterations (*maxNoSuccess*) or the user's target fitness (*targetFitness*) value has been reached.

A global increase in fitness means that a new best object ($o_b$) has been found. This also sets *lastSuccess* to the current iteration in order to make this fact known to the outer loop. Otherwise, it would break on $iter - lastSuccess >= maxNoSuccess$. The error of the new best object is then, at the end of the iteration, compared to the user's target fitness. If it is equal or better, the training ends and the newly found best object is returned.

The training function takes samples of the success of the training at a constant interval, $T$. This is used to calculate the mean success dynamically by using a Linear Time-Invariant system (LTI). This LTI is defined as:

$$pt1(y,u,t) = \begin{cases} u & \text{if } t = 0 \\ y + \dfrac{u-y}{t} & \text{otherwise.} \end{cases} \quad (4)$$

with $t = T$ in our case. The constant $T$ is user-defined and denotes the number of iterations between two samples. The *pt1(y, u, t)* function is called after a newly generated object has been tested for its fitness. If it is better than the worst object of the current population, we set $success = pt1(success, 1.0, T)$. But, only if this worst object has still iterations to live; if not, we set $u = -1.0$ in the call to *pt1(y, u, t)*, since replacing an already dead object cannot be counted as success.

This mean success is important during the generation of new objects, because it is used in order to calculate the *implicit gradient information*. These information are used to calculate a new object's parameters. Figure 1 shows schematically how a set of objects uses the implicit gradient information to move towards the optimum.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-357-5 

**global** $population, eliteSize, success, targetSuccess, successWeight, gradientWeight$
**local** $o_n, o_e, o_r, i_1, i_2, gradientSwitch, \Delta x, succcessRate, expvar, xlp$
$xlp \leftarrow 0.0$
$i_1 \leftarrow \mid \text{RANDOM}() \mod eliteSize - \text{RANDOM}() \mod eliteSize \mid$
$i_2 \leftarrow \text{RANDOM}() \mod population.length$
**if** $\text{OBJECT1ISBETTER}(population_{i_2}, population_{i_1})$ **then** $\text{SWAP}(i_1, i_2)$
$o_e \leftarrow population_{i_1}$
$o_r \leftarrow population_{i_2}$
$successRate \leftarrow success/targetSuccess - 1.0$
$gradientSwitch \leftarrow \text{RANDOM}() \mod 3$
**if** $gradientSwitch = 2$
**then** $\begin{cases} xlp \leftarrow (\sum_0^9 \text{FRANDOM}() - \sum_0^5 \text{FRANDOM}()) \cdot gradientWeight \\ \textbf{if } xlp > 0.0 \textbf{ then } xlp \leftarrow xlp \cdot 0.5 \\ xlp \leftarrow xlp \cdot \exp(gradientWeight \cdot successRate) \end{cases}$
$expvar \leftarrow \exp(\text{FRANDOM}() - \text{FRANDOM}())$
**for** $i \leftarrow 0$ **to** $o_n.\mathbf{p}.length$

$$\textbf{do} \begin{cases} \Delta x \leftarrow o_e.s_i \cdot \exp(successWeight \cdot successRate) \\ o_e.s_i \leftarrow \text{APPLYBOUNDSFROMEQUATION 6}(\Delta x) \\ \textbf{if } \text{FRANDOM}() < 0.5 \textbf{ then } \Delta x \leftarrow o_e.s_i \textbf{ else } \Delta x \leftarrow 0.5 \cdot (o_e.s_i + o_e.s_i) \\ \Delta x \leftarrow \Delta x \cdot expvar \\ o_n.s_i \leftarrow \text{APPLYBOUNDSFROMEQUATION 6}(\Delta x) \\ \Delta x \leftarrow o_n.s_i \cdot (\sum_0^4 \text{FRANDOM}() - \sum_0^4 \text{FRANDOM}()) \\ \textbf{if } gradientSwitch = 0 \\ \quad \textbf{then } \{\textbf{if } \text{RANDOM}() \mod 3 < 2 \textbf{ then } \Delta x \leftarrow \Delta x + o_e.p_i \textbf{ else } \Delta x \leftarrow \Delta x + o_r.p_i \\ \quad \textbf{else if } gradientSwitch = 1 \textbf{ then } \Delta x \leftarrow o_e.p_i \\ \quad \textbf{else if } gradientSwitch = 2 \\ \quad \textbf{then } \begin{cases} \Delta x \leftarrow \Delta x + o_e.p_i \\ \Delta x \leftarrow \Delta x + xlp \cdot (o_e.p_i - o_r.p_i) \end{cases} \\ o_n.p_i \leftarrow \Delta x \end{cases}$$

**return** $(o_n)$

Figure 2. The GENERATEOBJECT() function

In *GenerateObject()* as detailed in Figure 2, we pick a random elite object ($o_e$) and a random object of the whole population ($o_r$) in order to create the new offspring, $o_n$.

For this, we determine the influence of the implicit gradient information, *xlp*. It is used on a random basis with a probability of $p = \frac{1}{3}$. This prevents *GenerateObject()* from completely discarding objects with bad gradients. Discarding only happens because of an object's age. If it is used, we first create a custom uniform distribution by repeated calls to *frandom()*. We further modify *xlp* by $\exp(successRate \cdot gradientWeight)$.

The user is able to tune the influence of the implicit gradient information by modifying the Variable *gradientWeight*. Our experiments have shown that values in the range of $[1.0, 3.0]$ show great success. A value of 0.0 completely disables this feature. Similarly, the influence of the mean success can be disabled by setting *successWeight* to 0.0.

The actual delta (henceforth $\Delta x$) by which first the object's scatter and then its parameters are modified is first derived from the elite object's scatter as shown in Equation 5.

$$\Delta x = \begin{cases} 0.5 \cdot (o_e.s_i + o_r.s_i) & \text{if } frandom() < 0.5 \\ o_e.s_i \cdot \exp(successWeight \cdot success) & \text{otherwise.} \end{cases} \tag{5}$$

The bounds specified in Equation 6 are then enforced.

$$eamin \leq ebmin \cdot \mid o_e.p_i \mid \leq \Delta x \leq ebmax \cdot \mid o_e.p_i \mid \tag{6}$$

The three variables the define the limits have the following meanings: *eamin* is the absolute minimum for values and typically set to the smallest IEEE 32 Bit floating point number, i.e., $1 \cdot 10^{-32}$. *ebmin* is the relative minimum of scatter. It is user-tunable, but $(ebmin + 1.0) > 1.0$ must be true. *ebmax* is the relative maximum of scatter. It is also user-tunable. We suggest $ebmin < ebmax < 10.0$ per the results of our experiments.

The scatter is finally used to set the new object's parameters.

We detail the complete process of generating a new object in Figure 2.

## IV. DISCUSSION

The algorithm's two primary advantages over traditional evolutionary algorithms are its ability to use implicit gradient information and the dynamic density by which new objects spread out in the search space.

We call this dynamic attribute of the algorithm *reproduction probability density function*. It is controlled by the relationship of the two variables *success* and *targetSuccess*. If

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-357-5

$success > targetSuccess$, the spread of new objects is increased. It is decreased if the opposite holds true.

The function becomes obvious in *GenerateObject()*. Here, the current success rate influences not only the new object's scatter and parameter vector, but also those of the selected elite object. This way, objects move dynamically towards a minimum in the search space. Thus, the training algorithm does not only work by iteratively creating new object through mating and crossover, but also enables older elite objects to "learn" and improve.

An additional piece of information we can draw from the success of the different objects is an implicit gradient information. Implicitly, because it is available through the spread of the selected objects towards a minimum. It is most obviously in the assignment in Equation 7.

$$\Delta x = \Delta x + xlp \cdot (o_e.p_i - o_r.p_i) \tag{7}$$

However, using these information also harbors the danger of converging towards a local minimum instead of a global one. An evolutionary algorithm typically saves the user from this by its random crossover and mutation procedures which always carry a chance of escaping a local minimum. We also include this behavior since we enable this feature on a random basis via the *gradientSwitch* variable.

Preliminary tests have been conducted using 10 minutes mean wind speeds obtained from Germany's national weather service, DWD. We have compared the testing performance of our algorithm to that of Simulated Annealing [6]. In order to make the results comparable, both implementations use the same code base.

For the comparison, two independent ANNs have been identically configured and initialized. Both algorithms continuously trained their neural network with the same data. For each forecast, the ANNs have been fed with the last 12 10 minutes mean values in order to make use of the short-term memory the Elman ANN provides. The network was then used to forecast the next 10 minutes. The network's forecast was finally compared to the actual measurement provided by the national weather service in order to calculate the network's error.

During the training phase, our algorithm showed an almost constant training time with a variation of $\Delta t \leq 1s$. This substantiates that the population had a nearly identical "way to travel" to an optimum during re-training, doing so targeted based on the implicit gradient information. The Simulated Annealing algorithm, in contrast, produced widely varying training times. On average, our algorithm needed 5% of the time Simulated Annealing took.

In the day period of which Figure 3 shows a section, the mean error of the ANN our algorithm trained was 1.31, while the network the Simulated Annealing algorithm trained obtained a mean error value of 1.86.

Figure 3 depicts a representative section of a test run. One can observe the varying training time of the Simulated Annealing algorithm due to its completely stochastic nature, while our algorithm shows constant training time. The three large

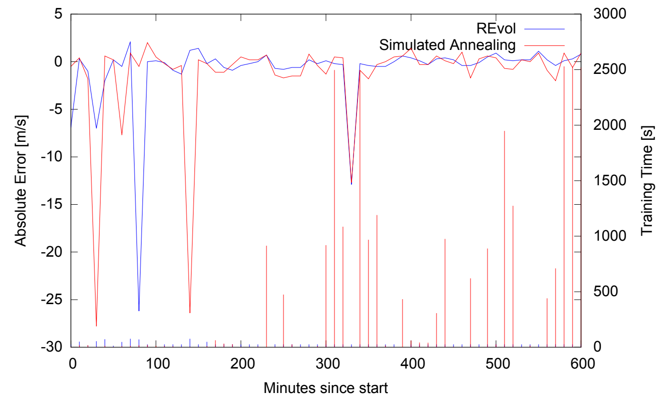


Figure 3. Absolute error and training duration of our algorithm ("REvol") and Simulated Annealing

spikes in the error values come from turbulences measured; interestingly, the network trained with our algorithm was able to forecast two of the three.

## V. Conclusion and Future Work

In this paper, we presented a training algorithm for ANNs that combines the strengths of evolutionary algorithms and deterministic ones. Our algorithm is able to include implicit gradient information into the reproduction process and allows training of already existing objects. It reduces the possibility of getting stuck in a local minimum, because it is based on paradigm of evolutionary algorithms that introduce randomness in order to escape local minima.

Due to these two features, we expect our algorithm to converge on a good minimum with a higher probability while still being able to escape a local minimum.

In the future, we will test our approach against other algorithms in terms of speed and convergence towards good minima. We will especially pay attention to Long-Short Term Memory approaches.

## VI. Acknowledgments

This paper has been created as part of a cooperative doctorate program between the TU Bergakademie Freiberg and Wilhelm Büchner Hochschule, Pfungstadt.

## References

[1] C. Potter, A. Archambault, and K. Westrick, "Building a smarter smart grid through better renewable energy information," in Power Systems Conference and Exposition, 2009. PSCE '09. IEEE/PES, March 2009, pp. 1–5.

[2] J. Elman, "Finding structure in time," Cognitive Science, vol. 14, no. 2, Jun. 1990, pp. 179–211.

[3] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," in IEEE International Conference on Neural Networks, 1993, pp. 586–591.

[4] J. Branke, "Evolutionary algorithms for neural network design and training," in Proceedings of the First Nordic Workshop on Genetic Algorithms and its Applications, 1995, pp. 145–163.

[5] I. Maqsood, M. Khan, and A. Abraham, "An ensemble of neural networks for weather forecasting," Neural Computing and Applications, vol. 13, no. 2, May 2004, pp. 112–122.

[6] S. Russel and P. Norvig, Artificial Intelligence: A Modern Approach, 3rd ed. Prentice Hall, 2010.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-357-5