Chapter 1
Generalized Neural Networks, Computational Differentiation, and Evolution*

David W. Juedes † Karthik Balakrishnan‡

Abstract

Backpropagation is a powerful and widely used procedure for training multilayer, feedforward artifical neural networks. The procedure can be seen as a special case of the reverse mode of computational differentiation. This connection between backpropagation and computational differentiation leads us to envision a scenario wherein neural networks can be trained by using gradient descent methods and computational differentiation tools like ADOL-C. The primary advantage offered by such an approach is the possibility of training networks consisting of heterogeneous functional units—a notion we refer to as *generalized neural networks*. This approach, in conjuction with *evolutionary algorithms*, can be used to produce near-optimal designs. This paper presents this approach in more detail and demonstrates its usefulness through simulation results.

1 Introduction

From pattern recognition to robotics, database retrieval to stock-market prediction, medical diagnosis to manufacturing and process control, artificial neural networks (ANNs) are finding widespread application [7]. Apart from their resemblance (though superficial) to the information-processing structures in the brain, their popularity can be attributed to a variety of factors—primarily their ability to elegantly capture *input-output associations*. Mathematically, ANNs perform a function, mapping the input space to the output space. What makes this model of computation especially interesting is that ANNs can be made to *learn* the desired input-output mapping by *training* the network on a sufficiently representative set of input-output examples. This set of examples is commonly referred to as the *training set*. The training process itself can be either *supervised* (wherein the desired output of the network corresponding to each input example is also provided) or *unsupervised* (the input examples are provided without any explicit specification of the outputs desired, and the network performs a sort of *clustering* of the examples based on some *similarity* or *difference* measure). The work reported in this paper deals only with the supervised class of algorithms, of which *backpropagation* (BP) is a popular example.

While training procedures like BP adapt the parameters of the ANN to realize desired input-output mappings, the actual mapping realized is limited by the following two factors (among others):

1. the *function* computed by each of the units in the network, and

*This work was supported in part by a Stocker Summer Fellowship.

†School of Electrical Engineering and Computer Science, Ohio University, Athens, Ohio 45701, juedes@ohiou.edu

‡Department of Computer Science, Iowa State University, Ames, Iowa 50011, balakris@cs.iastate.edu

2. the *connectivity pattern* of the units in the network, that is, the *network architecture*.

Current training algorithms typically work with *homogeneous networks*, wherein all the units compute the same function (e.g., threshold, sigmoid). The network architecture (number of hidden layers, number of units in the hidden layers, their connectivity etc.) is typically determined by brute-force trial and error or through heuristic techniques such as experience and rules of thumb. Thus, conventional ANN design techniques and training algorithms are grossly inadequate, especially if one is interested in compact or near-minimal networks for realizing complex mappings. Many researchers have argued this point, and there is considerable reason to believe that higher order networks may help realize mappings more efficiently and compactly than conventional ANNs [2].

In this paper, we examine neural networks that are inherently *heterogeneous*. These *generalized neural networks* differ from standard neural networks in the sense that they can incorporate a variety of activation functions. Here we show that these networks can compactly represent several well-studied problems. Moreover, we build on the connection between backpropagation and the reverse mode of computational differentiation to show that these networks can be trained quickly. (Others have made the connection between computational differentiation and neural networks. For example, Saarinen et al. [18] show that backpropagation is essentially the reverse mode of computational differentiation. In a related context, Yoshida [21] uses computational differentiation and two-dimensional conjugate gradient search to train neural networks.) Finally, we address the issue of network architecture design by demonstrating how evolutionary algorithms [5, 8] and computational differentiation can be used to build near-minimal networks.

2 ANN and BP

Conventional ANNs are highly interconnected networks of relatively simple computing elements. In the accepted parlance, the computing elements are referred to as *neurons* or *units*, while the interconnections are typically called *weights*. In addition to the weights, the units typically have another adaptable parameter associated with them, called the *threshold* or *bias*. The bias parameter allows the units to move or locally modify the shape of the activation function. In most cases, the units compute a weighted sum of their input signals and then transform this through the activation function into the output of the unit. Popular activation functions are the *linear threshold function* and the *sigmoid*, although *gaussian* functions also have their own following. These networks are assumed to be simplistic, organized structures, with well-defined input (*sensory*), hidden (*associative*), and output (*response*) layers, as shown in Figure 1.

The era of artificial neural computing began with the seminal work of McCulloch and Pitts [11], who observed that biological neurons behave in many ways like linear threshold units. Building on the work of McCulloch, Pitts, and others, Rosenblatt [14, 15, 16] defined a very simple learning device consisting of a single linear threshold unit built on top of additional hardware. These simple networks, called *perceptrons*, could be trained via a simple learning rule that roughly corresponds to gradient descent. Further, Rosenblatt [16] showed that perceptrons could be trained to perform every task in their limited repertoire. More precisely, Rosenblatt showed that if there is a setting of the weights and the threshold of the perceptron that solves a given problem, the learning rule is guaranteed to discover it. This is the famous *perceptron convergence theorem* [12]. Unfortunately, Minsky and Papert [12] showed that perceptrons are computationally limited and cannot compute nonlinearly separable functions. For example, a perceptron cannot compute the XOR function. Although

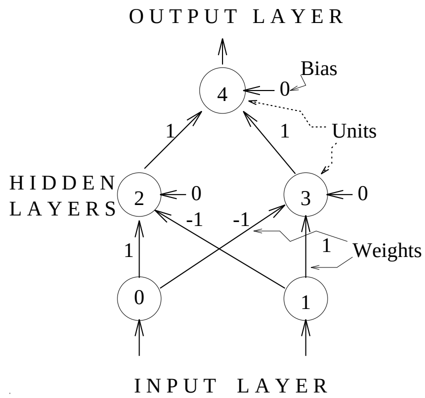

FIG. 1. *An artificial neural network*

multilayer perceptrons can compute *any* Boolean function, there is no analogous training rule for such networks because the internal threshold units are not differentiable and thus not amenable to gradient descent.

As noted by Rumelhart et al. [17], the training problem for multilayer neural networks can be solved by replacing the nondifferentiable threshold function by a differentiable function that approximates the threshold in the limit, for example, the *sigmoidal* function—$f(x) = \frac{1}{1+e^{-kx}}$. Such networks can be trained via a gradient descent procedure by adjusting the weights of the interconnections based on their contribution to the total network error. Rumelhart et al. [17] give a simple and efficient technique known as *backpropagation* for computing the partial derivatives of total error in the network with respect to the interconnection weights. (Note: BP was independently discovered earlier by Werbos [20].)

BP performs a gradient-descent in parameter space, minimizing an appropriate error function. Typically, the parameters are the *weights* and *thresholds* in the network. The conventional error function $\mathcal{E}$ is defined as the mean squared error over the output layer of the network taken over all the training patterns

$$\mathcal{E} = \frac{1}{2}\sum_{p}\sum_{i}(t_i^p - o_i^p)^2, \tag{1}$$

where t_i^p refers to the desired output of the ith output unit for the pth pattern and o_i^p refers to the output obtained at the ith output unit for the pth pattern.

In BP, there are two models in which weight-updates are performed: the *continuous model* where weights are updated after *every* input pattern presentation, and the *batch model* where weight changes are accumulated and weights updated after every epoch (all input patterns have been presented once). The continuous model is more efficient in terms of space, whereas the batch model is slightly more efficient in terms of time. Furthermore, the batch model makes weight changes based on the true gradient (since it accumulates the gradient information for all input patterns), whereas the continuous model uses only local information to make weight changes. (This local information is often referred to as the *instantaneous gradient.*) In our studies, we use the the continuous update model, and while doing so, we drop the pattern indicator p, since weight updates will take place after each pattern presentation.

With a continuous model assumption, the weight-update rule that minimizes $\mathcal{E}$ can be easily derived (refer Rumelhart et al. [17] for details of the derivation) and can be shown to be

$$\Delta w_{ij} = \eta \delta_j o_i. \tag{2}$$

In this equation, δ_j refers to the *instantaneous gradient* given by the expression

$$\delta_j = \begin{cases} (t_j - o_j).o'_j & \text{if j is an output unit} \\ (\sum_k w_{jk}\delta_k).o'_j & \text{if j is a hidden unit,} \end{cases} \tag{3}$$

where o'_j is the derivative of the activation function.

BP is guaranteed to find a local minimum of the error function, provided the weight changes are infinitesimal. The magnitude of weight changes (the step size) is determined by the *learning rate* η. Intuitively, large steps would help find a local minimum faster. However, if the steps are very large, the system could end up oscillating about a local minimum, maybe even never reaching the minimum. This is the *step size* problem. Rumelhart et al. [17] suggested a simple way out of this dilemma. They used a *momentum term* which factors in a fraction of the previous weight change into the current weight change. Such an arrangement makes the system take progressively larger steps along smooth error surfaces and successively smaller steps around local minima, thereby dampening the undesirable oscillations. The weight update rule using such a momentum term is

$$\Delta w_{ij}(t+1) = \eta \delta_j o_i + \alpha \Delta w_{ij}(t), \tag{4}$$

where $0 \leq \alpha \leq 1$ is a constant called the *momentum rate*.

3 Designing ANN

Even with BP-like training algorithms, designing appropriate ANN to achieve desired mappings is quite an art, since no sound procedure exists for determining either the network structure or the choice of the activation functions. This issue becomes especially critical if one is interested in near-minimal network sizes. For example, it can be shown that an ANN with two input units connected to a single sigmoid output unit cannot be trained to solve the XOR problem. For this task, at least one hidden unit is required. However, if the activation function of the output unit were chosen to be a cosine function, then it is easy to show that at least one set of weights exists that would make the network solve the XOR problem without the need for any hidden units. This is a trivial example of the ability to produce compact networks by the wise choice of the activation function.

It is well known in the neural network community that supplementing the inputs of an ANN with higher-order combinations of the inputs increases the capacity of the network [2] and the ability to learn geometrically invariant properties [4]. *Product units* [3], which compute a *weighted product* of their inputs rather than a weighted sum, fall into this category and have been shown to be of good use in a number of problems [10].

With these in mind, we have been exploring ANNs that contain units incorporating (possibly) different activation functions. We label such networks *generalized neural networks* (GNNs), more to differentiate them from conventional ANNs than to introduce any new nomenclature. In this work we constrain the choice of activation functions to ones that are differentiable. This would enable such networks to be trained using gradient descent techniques like BP. Further, the functions should also have *bounded* output (e.g., in the interval [-1, 1]) to prevent the weights from becoming unbounded themselves. Functions such as sigmoid, sine, cosine, tanh, and gaussian are suitable candidates for GNNs.

GNNs can be expected to produce compact mappings by drawing on the relative strengths of the different activation functions made available to the units. This capability is in keeping with the thesis of Uhr in [19], where a case is made for the development of *subnetworks* and *specialized circuits*, in order to build efficient ANNs. The basic claim is that specific subcircuits might be constructed to extract different features from the input, thereby allowing for better (more efficient) information representation and processing—a recurring feature of actual biological neural systems. For example, we might visualize a GNN with a layer of gaussian units, each localized to a different region of the input space, thereby extracting possibly independent features of the input, which are then combined in more complex ways by higher layers. Similarly, given the XOR problem, a compact GNN would exploit the strength of sinusoidal functions (sine or cosine) to construct a network consisting of a single output unit and no hidden units rather than using a sigmoid function and one or more hidden units.

4 GNN and ADOL-C

Training networks with arbitrary activation functions by using gradient descent methods presents a unique problem: How does one compute the partial derivatives in such a network? As formulated by Rumelhart et al. [17], BP only applies to *semi-linear functions*,[1] and not arbitrary composite functions. In place of BP, we propose the use of computational differentiation. In this section we show how ADOL-C[6] can be used to train GNN.

Consider an arbitrary GNN whose activation functions are composites of the standard functions like sigmoid, sine, cosine, and tanh. Such a network can be trained using ADOL-C by performing the following basic steps. (See [6] for more information and examples.)

1. Design a standard C function that evaluates the network and computes the error on a given input/output pair.

2. Convert each **float** or **double** variable in the C function to variable of the class **adouble** as defined in **adouble.h**.

3. Write a short piece of code that evaluates the **adouble** function on the given input/output pair and computes its gradient via the utilities in ADOL-C.

4. Use the gradient computed in (3) to adjust the weights in the network iteratively.

In addition to these four steps, the user may wish to use vector and matrix operations in ADOL-C (via **adoublev** and **adoublem** classes) to improve performance.

As an example, consider a single-node network with a sigmoidal activation function and n inputs. The code fragment in Figure 2 implements such a function in ADOL-C using vector operations. If we want the gradient of the function **network** with respect to the independent variables **wghts** and **theta**, we must perform some initial steps as required by ADOL-C. First, ADOL-C requires that the user begin "taping" the execution of **adouble** operations. Taping is initiated by the execution of the **trace_on** function and is transparent to the user. Second, ADOL-C requires that the user initialize and nominate the independent variables using the <<= operator. Third, ADOL-C expects the user to compute the function and nominate a dependent variable(s) via the >>= operator. Finally, ADOL-C requires that the user execute the **trace_off** function. At this point, the user can use one of several ADOL-C

[1] The techniques of Rumelhart et al. can be extended to arbitrary composite functions. This is precisely the reverse-mode of computational differentiation.

```
adouble network(adoublev &wghts,adoublev &inputs, adouble theta,
                adouble desired)
{
  adouble temp, wght_sum;
  adouble sum;
  wght_sum = wghts*inputs; /* Compute the dot product */
  temp = (1.0/(1.0+exp(-(wght_sum+theta))))-desired;
  return 0.5*temp*temp;
}
/********** Computing the gradient *************/
trace_on(1,0); /* Use tape #1 */
for (i=0;i<k;i++)
{
    wghts[i] <<=1.0; /* Nominate independent variables */
    inputs[i] <<=1.0;
}
desired <<=1.0;  theta <<= 1.0;
err = network(wghts,inputs,desired,theta); /* Compute the error */
err >>= err_out   /* err is the dependent variable */
trace_off();     /* Turn off tracing */
n = k+k+2        /* # of independent variables */
gradient(1,n,inps,grad); /* Call gradient */
```

FIG. 2. *An* **adouble** *function that computes the error in a sigmoid network*

utilities to compute the derivatives of the dependent variables with respect to the independent variables.

Once the function has been "taped," there are several ways to compute the appropriate derivatives. Since our function has a single result, we can use the built-in **gradient** routine to calculate the gradient of the taped function at any given input point. The use of this function is illustrated at the bottom of Figure 2. In the function call **gradient(1,n,inps,grad)**, the first parameter (**1**) refers to the tape number supplied in **trace_on**; the second parameter (**n**) refers to the number of independent variables; the parameter **inps** refers to an array of doubles that contains the new evaluation point; and the parameter **grad** refers to an array of doubles that is used for output. After the execution of the routine **gradient**, the array **grad** contains the partial derivatives of **err** (the dependent variable) with respect to the independent variables in the order in which the variables were nominated. In this case, **grad[0]** contains $\frac{\partial err}{\partial wght[0]}$ and **grad[2*k+1]** contains $\frac{\partial err}{\partial theta}$. Once the gradient has been computed, the weights in the network can be updated and the process can be repeated.

As our extended example illustrates, computing the gradient via ADOL-C is a reasonably straightforward technique and can be used to train GNN quite easily. The beauty of this approach is that we can compute the derivatives of any composite function (even dynamically) by using the same basic outline. We used these same techniques successfully in a number of simulations.

4.1 Automatic Differentiation vs. BP

As noted earlier, BP is essentially a special case of the reverse-mode of computational differentiation. So, it is natural to ask how well certain automatic differentiation tools perform in comparison with BP on standard neural networks. Here we answer this question for ADOL-C.

As a basis for comparison, we considered the performance of ADOL-C and BP when training a sequence of generic networks. As illustrated in Figure 3, our generic neural networks had n-inputs, k hidden layers, a single output, and sigmoidal activation functions. These networks were fully connected in the sense that each node in hidden layer k was connected to each node in hidden layer $k+1$. This network topology was chosen because it is easily generalizable and because it is straightforward to train via the BP algorithm as given in Rumelhart et al. [17].

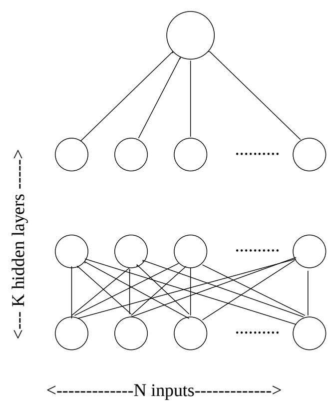

Fig. 3. *A generic test network topology*

Since we expect both BP and ADOL-C to train these networks equally well, our simulation centered only on the time required to compute derivatives. In our simulations we tested a variety of networks by varying the number of hidden layers and the number of inputs. Each simulation consisted of computing the partial derivatives via both BP and ADOL-C for some number of iterations.

We ran simulations under a variety of implementations. Our first implementation computed the gradient by taping a network evaluation function (that used only scalar operations) at each point and then calling the ADOL-C utility **reverse**. Using this technique, BP was on average 33.88 times faster than ADOL-C. Since it is well known that taping the function at each point incurs a large overhead, our second implementation removed this overhead by using the **gradient** utility as described in Section 4. Using this technique, BP was on average 27.89 times faster than ADOL-C. By introducing vector operations into our second implementation, ADOL-C was found to be only 20.8 times slower than BP.

The aforementioned timing results were performed by using compiler optimizations on all associated code. It is important to note that removing compiler optimizations greatly improved the performance of ADOL-C. By removing compiler optimizations in our third implementation, BP was only 10.27 times faster than ADOL-C.

While backpropagation outperforms ADOL-C, we note that this situation does not indicate a limitation in the power of automatic differentiation. Automatic differentiation can, in theory, rival the performance of backpropagation. Our timing comparisons give us some important clues about improving the performance of ADOL-C towards this goal. For example, the significant difference in performance between the optimized and unoptimized versions of our simulations suggests that one problem may lie in the inability of C++ to optimize user defined data types. These issues clearly need further investigation.

5 Network Architectures and GAs

In the earlier sections, we have made a case for GNNs by showing its increased functional power over traditional homogeneous networks. In the preceding section we demonstrated a technique based on the computational differentiation tool ADOL-C that allows us to perform the equivalent of BP training without explicit knowledge of the *exact symbolic* form of the derivatives of the activation functions. However, the problem of determining good network architectures still remains an elusive issue.

Some heuristic approaches in the literature allow one to *incrementally* construct networks by adding one or more units at a time. Popular examples of such *constructive algorithms* are tower, tiling, upstart, and cascade-correlation [13]. The first three algorithms typically are used for constructing multilayer perceptron networks, while cascade-correlation is used with networks of sigmoid units. Although promising in theory, the performance of these algorithms is critically dependent on the choice of the activation funtion (e.g., threshold or sigmoid) and the training procedure used (e.g., perceptron, BP). These biases need to be explored in more detail if the methods are to be used on real-world problems. (A systematic evaluation and comparision of these and other approaches is already underway [13].)

As an alternative, Leerink et al., [10] and others have used combinations of global and local search techniques. Such approaches use a procedure like *simulated annealing* (SA) or *random restarts* [9] to perform a global search for network architectures, which are then locally trained using BP. The effectiveness of the global searches in this context is bounded by the dimension of the search space and the density of the solution points. Thus, if the problem on hand is characterized by a large search space with sparsely populated solutions — the random searches become highly ineffective.

Evolutionary approaches such as *genetic algorithms* (GAs), have been shown to produce near-optimal results in vast, complex, and multimodal search spaces [8, 5]. They are thus natural candidates for searching the space of network architectures [1]. These algorithms are models of processes that appear to be at work in biological evolution. Such systems work with populations of *genotypes*. (In our application, a genotype is a *specification* of an ANN.) Each genotype is itself a collection of structures (called *genes*), typically arranged in some string-like fashion (akin to the chromosomal strings in biological systems). The genotypes are transformed into *phenotypes* (actual ANN) through a process referred to as *decoding*. Once decoded, the phenotype (ANN) may be evaluated on the task at hand with respect to suitable performance measures (e.g., accuracy, size, cost, training speed). The evaluation of the phenotype produces a *fitness label* that can be used to rank the corresponding genotypes, as shown in Figure 4.

The evolutionary procedure works by preferentially selecting genotypes for reproduction based on their fitness and introducing variety into the population through genetic operators such as mutation, crossover and inversion. This process, continued over many generations, results in the populations gradually evolving toward genotypes that correspond to high-fitness phenotypes.

The following section describes some of the results of our simulations with GNN.

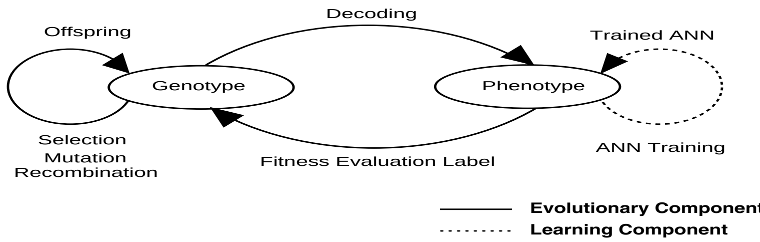

FIG. 4. *Process of evolutionary design of neural architectures*

6 Simulation Results

The ability of GNNs to represent mappings through compact networks is most easily illustrated in single-node networks. Consider, for example, the parity problem.[2] While it is well known [12] that a network of a single sigmoidal or threshold unit cannot represent the parity problem, it is straightforward to design single-node sine or cosine networks to solve parity. (See, for example, Figure 5.) Moreover, it is relatively straightforward to prove that a network of a single sine gate can solve odd-parity on all inputs of odd size and even-parity on all inputs of even size. Similarly, a network of a single cosine gate can solve even-parity on all inputs of odd size and odd-parity on all inputs of even size. Unfortunately, no single-node sine gate network with an even number of inputs can solve the odd-parity problem and vice-versa. A similar situation occurs for single-node cosine gates. This situation can be remedied by adding a bias factor. To see this, notice that a network of a single sine or cosine gate with a bias θ (i.e., the activation function is $\sin(\sum_j w_j i_j + \theta)$) can solve both even- and odd-parity on all inputs.

Since single-node sine networks can solve parity, the next natural question to ask is whether these networks can *learn* to solve parity. As our simulations demonstrate, the answer to this question is yes.

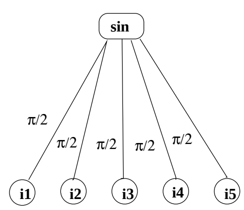

FIG. 5. *Our solution to the parity problem on five inputs*

Our initial simulations consisted of training single-node sin and $\sin + \theta$ networks on the parity problem for inputs of size 2 through 7. Our single node networks were given initial weights of 1.0 and an initial bias of 1.0. These networks were trained against a set that contained all bipolar (-1,1) inputs of each size. As in Figure 5, our training algorithm found weights that correctly solved the parity problem in each case.

6.1 Multi-layer Networks

As shown in the preceding section, individual units are themselves powerful entities, since they can carve out regions of specific shapes. Multilayer GNNs can exploit this property with

[2] Here we assume that each input to the network is either -1 or 1 and that 1 represents TRUE and -1 represents FALSE. The *even parity* problem is to determine whether the number of 1's in the input is even, and the *odd parity* problem is to determine whether the number of 1's in the input is odd.

structures wherein lower layers are composed of units with different activation functions and higher layers combine the results of lower layers in complex ways. This approach has two fundamental problems. The first problem lies in determining the appropriate activation functions. The second problem lies in determining the appropriate way to interconnect these units to form networks capable of implementing the desired mapping. For reasons mentioned earlier, we resort to evolutionary algorithms to determine the network structure and the unit activation functions and rely on ADOL-C to produce the desired weights through training.

We used the genetic representation shown in Figure 6. Here the genotype is a sequence

Connectivity Matrix

From Unit

To Unit	0	1	2	3	4	5	bias
2	1	-1					1
3	1	2	-2				-5
4	2	0	-3	1			1
5	0	1	-1	2			-1

Genotype

Spherical 1 -1 0 0 0 0 1	Cosine 1 2 -2 0 0 0 -5	Sine 2 0 -3 1 0 0 1	Sigmoid 0 1 -1 2 0 0 -1

Network (Phenotype)

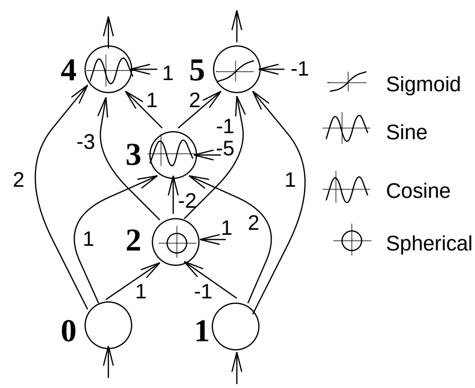

FIG. 6. *The genotypic representation used*

of genes that each encode the activation function and input connectivity of one unit of the network.

Assume for the moment that we have a network with **max** units, **n** inputs, and **m** outputs. In our encoding, the input units are labeled with 0 through $n-1$, and the output units are labeled with numbers from **max -1** to **max -m**. The remaining units form the pool of hidden units. Here we allow only feedforward connections. As an example of this genetic representation, consider Figure 6. In this example, the network has two input units labeled 0 and 1, two outputs labeled 4 and 5, and two hidden units labeled 2 and 3. In this network, unit 2 computes the *spherical* function, and units 3, 4, and 5 compute the cosine, sine, and sigmoid functions, respectively.

We used the above genetic representation in our simulations. Each simulation run started off with a random initial population of 100 individuals, called generation 0. The fitness of each genotype was determined by decoding it into the phenotype ANN and training the resulting network by using the procedure outlined in Section 4. The resulting accuracy of the network after 100 presentations of the training set was taken to be the fitness of the corresponding genotype. Higher-fitness individuals were selected with proportionally higher probability to act as parents. Offspring were produced from the parents by randomly choosing genes from the two parents—a process known as *crossover*. In addition, with low probability, random genes were altered by random amounts—the artificial equivalent of the *mutation* process. (See Figure 7 for an illustration of the uniform crossover and the mutation operators.) The offsprings then replaced the parents, thus producing the next generation. This process was continued over 100 generations.

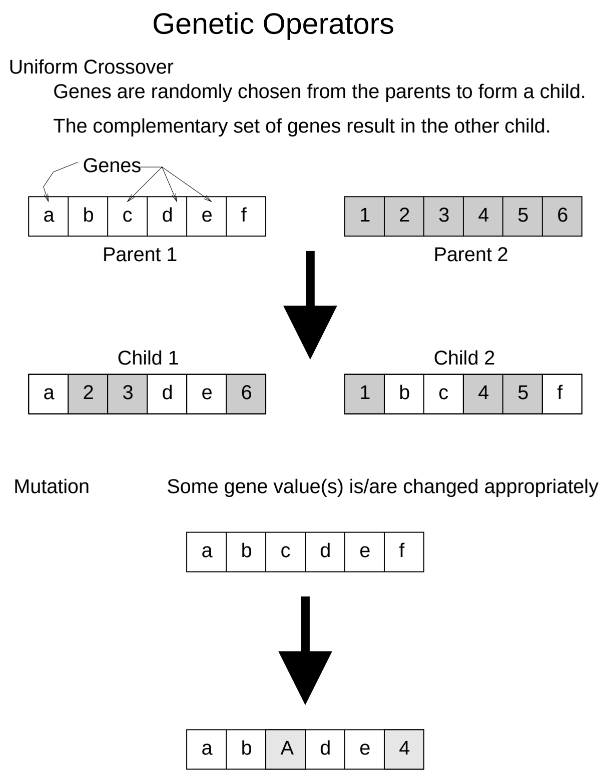

FIG. 7. *The genetic operators — uniform crossover and mutation*

Let us illustrate the benefits of this approach with a quick example. The *2-circles* problem, shown in Figure 8, consists of a 2-dimensional real input space containing two concentric circles. The training set consists of 64 patterns such that if an input point lies in the region between the two circles, the corresponding output generated by the ANN is desired to be a $+1$, while if the point happens to lie inside the smaller circle, the network is expected to produce a -1. This data set has been used extensively in the literature because it is small, yet non-trivial. Since the decision boundaries for this problem are circular, we would in principle, need a juxtaposition of an infinite number of threshold units to approximate this boundary. However, in practice, a finite (though large) number of units suffice because the training set is finite, and hence the training points themselves encode only an approximation of the circular decision boundary.

The ability of the network to correctly classify hitherto unseen examples (examples not in the training set) is called its *generalization* performance. Clearly, to obtain perfect generalization on the 2-circles problem, the decision boundary has to be perfectly circular. This can be achieved by allowing higher-order activation functions, namely an approximate *spherical* function. One such function is $t_i = \sum_j (w_{ji}.o_j)^2 - w_{0i}^2$, where the units identified

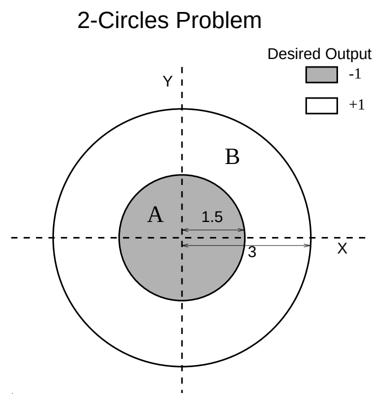

FIG. 8. *The 2-circles problem*

by j connect to unit i, and w_{0i} is the threshold or bias of unit i. This activation function defines the Cartesian equation for an n-hypersphere centered at the origin, with radius w_{0i}. This activation function can be used in our ANNs with the understanding that the unit i will produce an output $o_i = 1$ if $t_i < 0$ (i.e., the point lies inside the hypersphere) and $o_i = -1$ otherwise.

Though this formulation is intuitively appealing, it is impossible to perform gradient-based training for such units because the output function o_i, as defined above, is not differentiable (o_i is a threshold function of t_i and hence has a discontinuity at 0). However, this problem can be resolved by approximating the thresholding action by a sigmoid function with an appropriately steep slope. For example, $o_i = \frac{2}{(1+e^{(100\times t_i)})} - 1$ is a sufficiently close approximation of the desired function and is differentiable. With these spherical units at their disposal, GNNs might be expected to easily address the 2-circles problem.

In the first set of experiments, evolution was used to determine a solution ANN with at most three sigmoid units (one output and two hidden). The best network produced by evolution attained an accuracy of only 92%. Clearly, the number of sigmoid units made available was not enough to realize the 2-circles problem. Rather than search for larger sigmoid networks, the next set of simulations aimed to evolve GNNs by permitting the units to choose from among sigmoid, sine, cosine, and spherical functions. Given this choice, the best network discovered by evolution is shown in Figure 9.

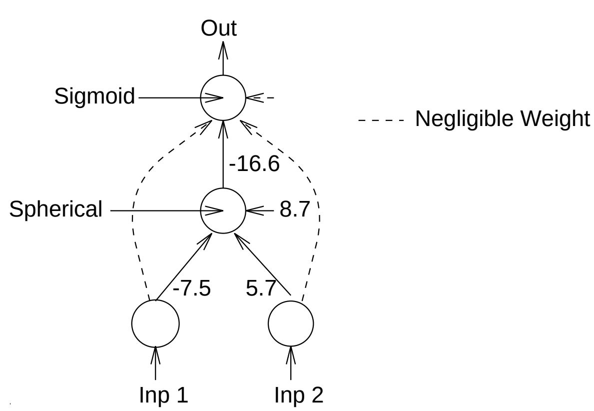

FIG. 9. *A network evolved for 2-circles problem*

As can be observed, the network uses a spherical unit to carve out the circular region corresponding to the inner circle in Figure 8. However, since we have defined the spherical function to produce a 1 on points *inside* the circle, the output of this unit is the *exact complement* of the outputs desired. Hence, evolution uses a sigmoid output unit to simply *flip* the output generated by the spherical hidden unit, as can be inferred from the large negative link between the two units in Figure 9. This network results in a 100% classification accuracy.

The above was just a simple illustration of the range of possibilities. One could easily build a library of such specific activation functions, *without* the need to know the exact symbolic forms of their derivatives. In addition, as in the case of the spherical function we just defined, individual units are not limited to computing one function; they could possibly perform *compositions* of functions. This aspect gives our approach a sizeable advantage over conventional BP, since BP would require one to explicitly program the derivative functions in addition to formulating the derivatives for specific forms of function compositions. With a computational differentiation tool none of this is required of the user; the system performs these tasks automatically.

Given this advantage that GNN's can exploit, evolution can be expected to construct appropriate structures for a variety of problems. In particular, with appropriate advances in multi-objective optimization using evolutionary techniques, this procedure might reliably be used to design highly fit, yet compact GNN.

7 Conclusion

The training of artificial neural networks presents both a challenge and an opportunity for work in computational differentiation. The problem at hand, namely, to compute the partial derivatives of the error in the network with respect to the interconnection weights, is well suited to the reverse-mode of computational differentiation. The challenge here is to rival the efficiency of the standard backpropagation algorithm.

Ideally, computational differentiation can match the efficiency of backpropagation. A real opportunity for computational differentiation lies in the training of neural networks where the backpropagation algorithm is difficult to formulate, as is the case in generalized neural networks. As shown in this paper, such networks have the potential to be qualitatively more appealing than their classical counterparts.

Given the widespread need for constructing compact, efficient, and low cost ANN, we expect that continued work on computational differentiation, generalized neural networks, and evolutionary network design approaches will lead to extremely fruitful ventures in the near future.

Acknowledgments

We thank Andreas Griewank, Jean Utke, Chris Bischof, and the referees for their extremely helpful comments.

References

[1] K. Balakrishnan and V. Honavar, *Evolutionary design of neural architectures — a preliminary taxonomy and guide to literature*, Tech. Rep. CS TR 95-01, Department of Computer Science, Iowa State University, Ames, IA - 50011, January 1995.

[2] T. Cover, *Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition*, IEEE Transactions on Electronic Computers, 14 (1965), pp. 326–334.

[3] R. Durbin and D. Rumelhart, *Product units: A computationally powerful and biologically plausible extension to backpropagation networks*, Neural Computation, 1 (1989), pp. 133–142.

[4] C. Giles and T. Maxwell, *Learning, invariance, and generalization in high-order neural networks*, Applied Optics, 26 (1987), pp. 4972–4978.

[5] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, Massachutsetts, 1989.

[6] A. Griewank, D. Juedes, and J. Utke, *ADOL-C, A package for the automatic differentiation of algorithms written in C/C++*, ACM Trans. Math. Software, (to appear). Also appeared as Preprint MCS–P180–1190, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., November 1990.

[7] R. Hecht-Nielsen, *Neurocomputing*, Addison-Wesley, Reading, MA, 1990.

[8] J. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975.

[9] N. Karmarkar and R. Karp, *The differencing method of set partitioning*, Tech. Rep. UCB/CSD 82/113, University of California, Berkeley, Computer Science Division, 1982.

[10] L. Leerink, C. Giles, B. Horne, and M. Jabri, *Learning with product units*, in Advances in Neural Information Processing Systems 7, MIT Press, 1995, pp. 537–544.

[11] W. S. McCulloch and W. Pitts, *A logical calculus of ideas immanent in nervous activity*, Bulletin of Mathematical Biophysics, 5 (1943), pp. 115–133.

[12] M. Minsky and S. Papert, *Perceptrons*, MIT Press, 1969.

[13] R. Parekh, J. Yang, and V. Honavar, *Constructive neural network learning algorithms for multi-category pattern classification*, Tech. Rep. ISU CS TR 95-15a, Iowa State University,, Department of Computer Science, 1995.

[14] F. Rosenblatt, *The perceptron: A probabilistic model for information storage and organization in the brain*, Psychological Review, 65 (1958), pp. 386–408.

[15] ———, *Perceptron simulation experiments*, Proceedings of the IRE, 48 (1960), pp. 301–309.

[16] ———, *Principles of Neurodynamics*, Spartan, New York, 1962.

[17] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning internal representations by error-propagation*, in Parallel Distributed Processing, D. E. Rumelhart and R. J. McClelland, eds., MIT Press, Cambridge, MA, 1986, ch. 8.

[18] S. Saarinen, R. Bramley, and G. Cybenko, *Neural networks, backpropagation, and automatic differentiation*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, Penn., 1991, pp. 31–42.

[19] L. Uhr, *Digital and analog microcircuit and sub-net structures for connectionist networks*, in Artificial Intelligence and Neural Networks: Steps Toward Principled Integration, V. Honavar and L. Uhr, eds., Academic Press, 1994, ch. XV, pp. 341–370.

[20] P. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, PhD thesis, Committee on Appl. Math., Harvard Univ., Cambridge, Mass., November 1974.

[21] T. Yoshida, *Rapid learning method for multilayered neural networks using two dimensional conjugate gradient search*, Journal of Information Processing, 15 (1992), pp. 79–86.