Accepted for presentation at the Seventh International Conference on Neural Information Proccessing, ICONIP'2000, Taejon, Korea, November 14-18 2000

A Comparison of Different Neural Methods for Solving Iterative Roots

Lars Kindermann[1] kindermann@forwiss.de www.forwiss.de/kindermann

Achim Lewandowski[1] lewandowski@forwiss.de www.forwiss.de/lewandowski

Peter Protzel[2] peter.protzel@e-technik.tu-chemnitz.de www.infotech.tu-chemnitz.de/~proaut

[1]FORWISS - The Bavarian Research Center for Knowledge-Based Systems
University of Erlangen, Am Weichselgarten 7, 91058 Erlangen, Germany

[2]Dept. of Electrical Engineering and Information Technology
Chemnitz University of Technology, 09107 Chemnitz, Germany

Abstract

Finding iterative roots is the inverse problem of iteration. Iteration itself plays a major role in numerous theories and applications. So far it is hardly realized how many problems can be related to its counterpart. This may be due to the difficulty of the mathematics involved: There are no standard methods available for computing these fractional iterations.
Previously we have shown how neural networks can be utilized to perform calculations of iterative roots by adding a weight coupling mechanism to backpropagation learning. Here we show that an easier implementation of this functionality can be achieved by a simple weight copy function. Introducing second order methods like quasi Newton learning on the other hand can significantly reduce training times and improve the reliability of the method. It also overcomes some limitations in the complexity of the problems the method can be applied to.

1 Iterative Roots

Iterative roots are the extension of the roots of numbers to the domain of function spaces. Thus they are defined in terms of *functional equations*:

Definition:

Given an arbitrary function $F(x): \mathbb{R}^n \to \mathbb{R}^n$, a solution $f(x)$ of the equation

$$f(f(x)) \equiv F(x)$$

is called a *functional* or *iterative root* of F.

In general, for the equation

$$f^k(x) = f(f(\ldots f(x)\ldots)) \equiv F(x)$$

the function $f = F^{1/k}$ is a *k-th iterative root* of F.

For linear transformations, F can be written as a square *n x n* matrix. Thus finding the iterative root *f* means solving the matrix equation $f \bullet f = F$ or $f^{n} = F$

This extends the notation of iterates or powers of functions from the known integer exponents like F^{-1} donating the inverse, F^0 the identity, and $F^{n \in \mathbb{R}}$ the n-th iteration of F, to *fractional iterations.*

Examples:

$$F(x) = ax \quad \Rightarrow \quad f(x) = F^{1/k} = \sqrt[k]{a}\, x$$

$$F(x) = x + b \quad \Rightarrow \quad f(x) = F^{1/k} = x + \frac{b}{k}$$

$$F(x) = x^2 \quad \Rightarrow \quad f(x) = F^{1/2} = |x|^{\sqrt{2}}$$

$$F\vec{x} = \begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix} \vec{x} \quad \Rightarrow \quad F^{1/k}\vec{x} = \begin{bmatrix} \cos\frac{\varphi}{k} & -\sin\frac{\varphi}{k} \\ \sin\frac{\varphi}{k} & \cos\frac{\varphi}{k} \end{bmatrix} \vec{x}$$

It is by no means easy to find the roots of a given function, to prove the existence or nonexistence of a root or to show the uniqueness of a solution. "Even ... the behaviour of iterative roots may sharply contrast with what one might expect"[1]. At least it can be shown that iterative roots of all orders exist for all continuous and strictly increasing real valued functions.

If a transformation $F_t(x)$ describes the development of a system *x* in some fixed amount of time *t*, there should exist any root of *F*, which models the variation of *x* in a fraction of the time: $F_t^{1/k} = F_{t/k}$ This means, data sampled at regular intervals from a continuously evolving system can be interpolated to yield a higher time resolution in an more exact and natural way than other interpolation methods can achieve.

If *F* is given only implicitly as a table of input - output data pairs (*x*,*y*), finding the iterative root becomes part of a regression problem: Finding the function *f* which fits $y = f^k(x)$ in an optimal way.

Applications

There are many problems, both from theory and practical applications, which can be related to solving functional roots.

Chaos Theory
Iterated functions play a key role in chaos theory. The logistic equation $x_{n+1} = \lambda x_n(1 - x_n)$ generates chaotic sequences x_n for $\lambda > 3.57$ and the famous Mandelbrot-set results from the same iteration, only with complex-valued λ [2]. Iterative roots may allow finding the basic generative functions from chaotic data.

Modelling Industrial Processes
Many production lines consist of a number of identical steps in a row. In steel mills you can find a number of stands which consecutively roll steel plates into thin sheets [3], in paper machines there are up to 30 heaters which are used to dry the pulp. Often only data from the incoming and outgoing material can be derived, but it is important to know how it evolves in between. If there is a data driven model of the whole line, a functional root of it will model the parts and thus yield information not accessible otherwise.

Financial forecasting
Often in historical records only monthly or weekly data is available and can be used to train a neural model. But if daily predictions are requested, it can be assumed, that a week consists of seven (or perhaps five trading-) days. The seventh root of the week to week model could be used for daily predictions then.

2 Computing Iterative Roots with Neural Networks

Facing a lack of standard methods, formal, numerical or data driven, to compute iterative roots, we were able to show that neural networks can successfully be applied to this problem [5].

Given some data set $(\bar{x}, \bar{y})\ \bar{x}, \bar{y} \in \mathbb{R}^n$, finding a function $F(\bar{x})$ which tries to fit $\bar{y} = F(\bar{x})$ is a standard problem for neural networks: Function approximation. MLPs with one or more hidden layers are widely used for this task. If we know that F is in fact a chain of similar steps, this can be mapped to a special network topology:

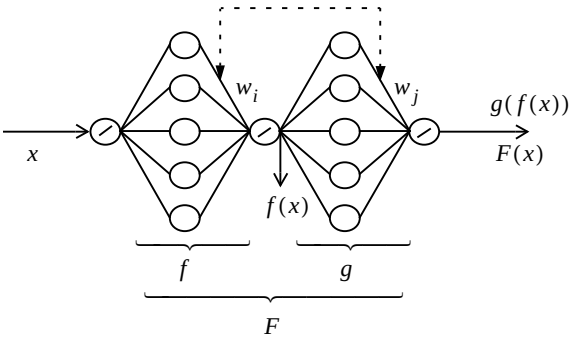

Figure 1: A MLP with this repetitive topology splits the net function F in two separable parts, f and g. Coupling all corresponding weights forces f and g to become identical and thus a functional root of F.

Such a network could be interpreted as a chain of functions $F(x) = g(f(x))$, where the first subnet represents f and the second one g. If we keep all the weights the same between the parts of the net, f will be identical to g, and thus be a functional root of F. This is similar to backpropagation through time networks [4], which are used to unfold recurrent networks. But some modification has to be applied to the learning algorithm in order to equalize the corresponding weights.

Hard Weight sharing

Backpropagation can easily be modified in a way, that the same weights are shared between several connections ($w_i = w_j$), they are initialized with the same value, and the deltas resulting from the learning rule are summed up and applied to all shared weights.

Soft Weight coupling

Because of better training performance we modified this algorithm by starting with different initializations and force the weights to approach each other slowly, by adding an additional term to the standard delta-rule [5] for weight-update:

$$\delta w_i = \delta w_{i\ \text{backprop}} + \alpha(w_j - w_i)$$

where w_j is the current value of the corresponding weight within the other layer and α an user definable parameter, we call *coupling factor*.

This is equivalent to the addition of a penalty-term, the sum of the squared differences of corresponding weights, to the error function, but easier to implement We found it most successful to cut the training process in two steps. First, the network is trained with standard backpropagation without weight coupling ($\alpha = 0$) until it reproduces $F(x)$ with appropriate accuracy. Then the factor α is slowly increased while keeping the whole net approximating F continuously.

Setting $\alpha = 1$ immediately often results in a failure of the net to learn F, especially when working with higher order iterations, i.e. using networks with many single neuron layers. These bottlenecks strongly inhibit the error backpropagation and the fewer degrees of freedom implied by that “hard” weight coupling seem to disturb the learning ability additionally.

Weight Copy

Weight copying is an even simpler version of hard weight sharing. Only the weights of the last layer are trained by backpropagation, and are copied back to the layers before. This allows for a very simple implementation in many tools which won’t allow incorporating a shared weights scheme.

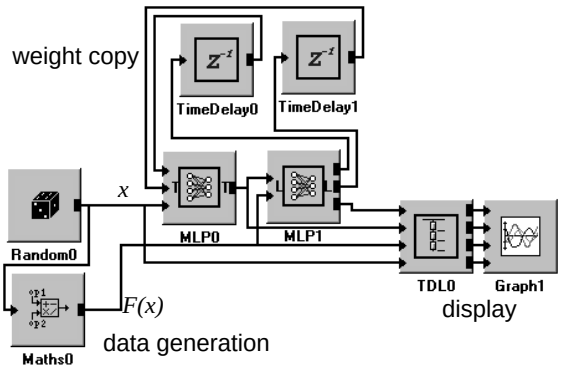

Figure 2: Implementation of the weight copy method in the commercial ECANSE programming environment.

3 Quasi Newton Learning

Because of the poor performance of these methods in case of higher order roots or complicated functions we tried to incorporate better learning rules to the problem. By designing the learning rule directly for iterated networks, we achieved a much better performance, reliability and scalability. Second order learning rules have been also proven to outperform backpropagation and modifications like rprop and quickprop in most cases and so they do here.

The algorithm is illustrated in the case of one single input. We used a simple representation for f with sigmoid activation functions ($\phi(x) = \tanh(x)$ could be used as well) for the hidden layer and linear outputs so that with $X_i = f^i(x)$, X_{i+1} can be recursively written as

$$X_{i+1} = \sum_{j=1}^{H} c_j \phi(a_j X_i + b_j)$$

If we try to find the k-th iterative root, the error function is given by

$$E = \sum_{d=1}^{D} (X_{k,d} - Y_d)^2,$$

with $X_{k,d}$ denoting the net output and Y_d the true output for the d-th example in the data set.

We used the Broyden-Fletcher-Goldfarb-Shanno (BFGS) procedure described in [7]. One needs the partial derivatives of E with respect to the weights a_h. b_h and c_h, for example

$$\frac{\partial E}{\partial a_h} = \sum_{d=1}^{D} 2(X_{k,d} - Y_d)\frac{\partial X_{k,d}}{\partial a_h}$$

Usual backpropagation algorithms can't be used without modifications, as the same weights appear in every subnet.

Because the input X_0 doesn't depend on the weights, which means that the partial derivatives

$$\frac{\partial X_0}{\partial a_j} = 0, \frac{\partial X_0}{\partial b_j} = 0 \text{ and } \frac{\partial X_0}{\partial c_j} = 0$$

vanish, it is possible to give the recursive formulas (δ_{jh} denotes the Kronecker-symbol):

$$\frac{\partial X_{i+1}}{\partial a_h} = \sum_{j=1}^{H} c_j \phi'(a_j X_i + b_j)\left(\delta_{jh} X_i + a_j \frac{\partial X_i}{\partial a_h}\right)$$

$$\frac{\partial X_{i+1}}{\partial b_h} = \sum_{j=1}^{H} c_j \phi'(a_j X_i + b_j)\left(\delta_{jh} + a_j \frac{\partial X_i}{\partial b_h}\right)$$

$$\frac{\partial X_{i+1}}{\partial c_h} = \sum_{j=1}^{H} \left(\delta_{jh}\phi(a_j X_i + b_j) + c_j \phi'(a_j X_i + b_j)\left(a_j \frac{\partial}{\partial c_h} X_i\right)\right)$$

Repetitive use of these formulas allows us to compute the necessary components

$$\frac{\partial X_k}{\partial a_h}, \frac{\partial X_k}{\partial b_h} \text{ and } \frac{\partial X_k}{\partial c_h}.$$

The multi-dimensional case differs only in so far as the needed derivatives get more complicated to write down because a derivative $\frac{\partial X_{i+1,t}}{\partial a_h}$ does not depend only on $\frac{\partial X_{i,t}}{\partial a_h}$, but also on all $\frac{\partial X_{i,s}}{\partial a_h}$ with $s \neq t$. Nevertheless the same strategy as before can be applied.

4 Examples

We present two examples here, a simple one for which the weight copy method is sufficient and another one which becomes solvable in practice only by introduction of the quasi Newton method.

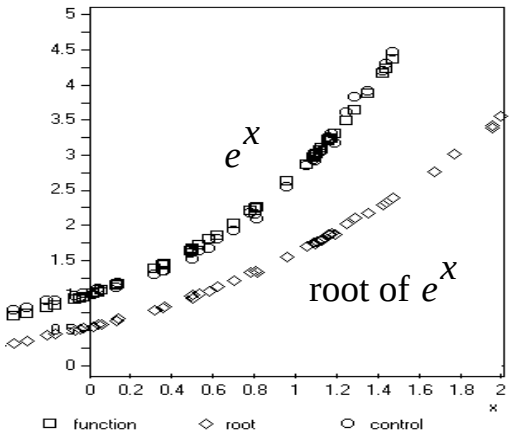

Figure 3: The root of e^x, computed with the ECANSE worksheet of figure 2. It is easy to find even with this simple method but seems to be a very hard problem to solve analytically.

Rotating points in the 2-D plane by the angel φ can be represented by a matrix. If we know that this happens in k steps, we can „guess" how the resulting matrix (the k-th root) looks like:

$$\begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix}^{\frac{1}{k}} = \begin{bmatrix} \cos(\varphi/k) & -\sin(\varphi/k) \\ \sin(\varphi/k) & \cos(\varphi/k) \end{bmatrix}$$

Figure 4 shows how a network can find this solution and several others too which are not as easy to „guess". It was trained with the quasi Newton method.

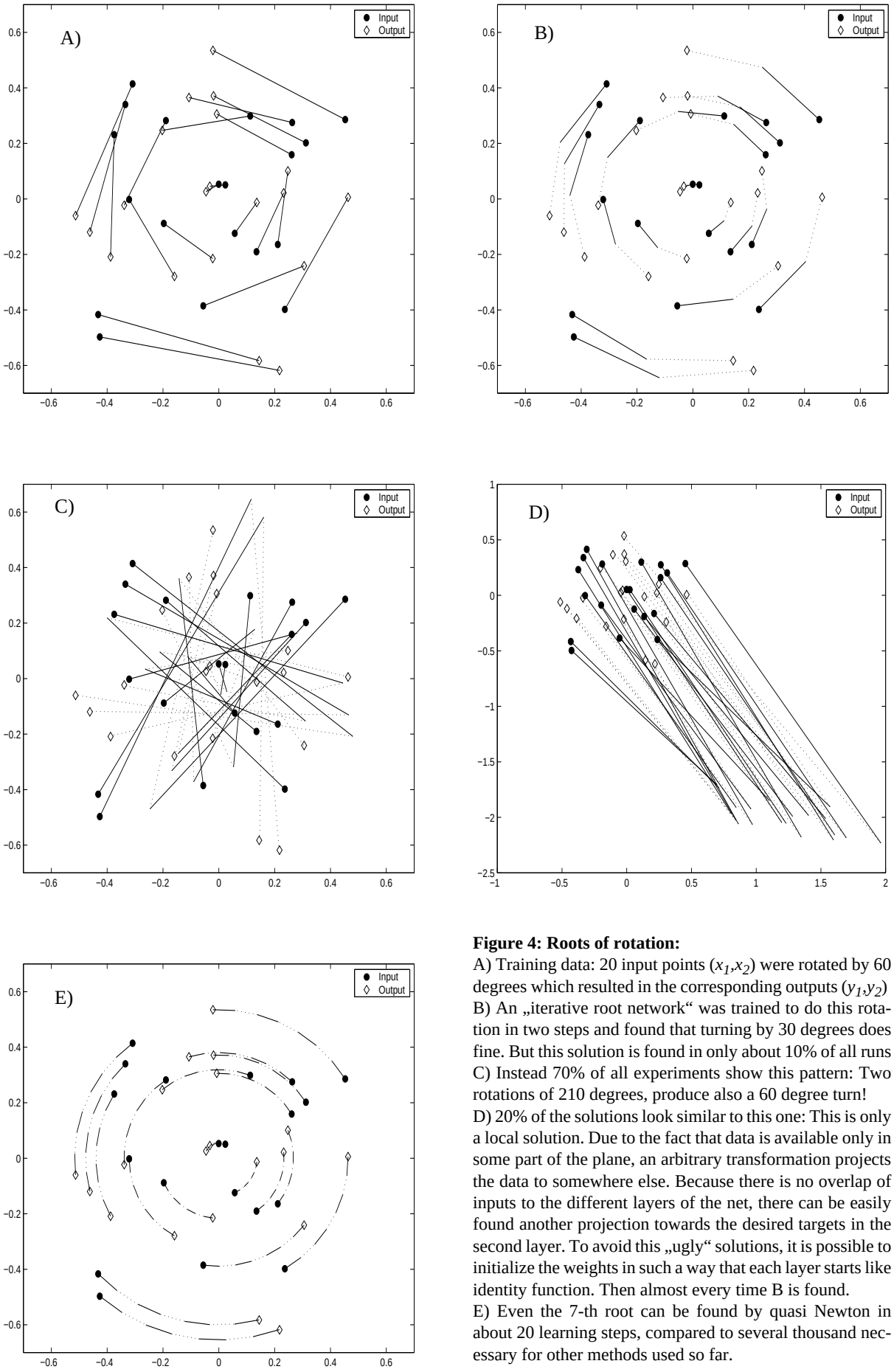

Figure 4: Roots of rotation:
A) Training data: 20 input points (x_1,x_2) were rotated by 60 degrees which resulted in the corresponding outputs (y_1,y_2)
B) An „iterative root network“ was trained to do this rotation in two steps and found that turning by 30 degrees does fine. But this solution is found in only about 10% of all runs
C) Instead 70% of all experiments show this pattern: Two rotations of 210 degrees, produce also a 60 degree turn!
D) 20% of the solutions look similar to this one: This is only a local solution. Due to the fact that data is available only in some part of the plane, an arbitrary transformation projects the data to somewhere else. Because there is no overlap of inputs to the different layers of the net, there can be easily found another projection towards the desired targets in the second layer. To avoid this „ugly“ solutions, it is possible to initialize the weights in such a way that each layer starts like identity function. Then almost every time B is found.
E) Even the 7-th root can be found by quasi Newton in about 20 learning steps, compared to several thousand necessary for other methods used so far.

5 Conclusions

The two new methods for computing iterative roots mean a significant advantage for applications:

The simple weight copy scheme allows people who don't want to do the programming which is necessary for more sophisticated methods, to compute iterative roots of simple functions in a numerical way. This can be a valuable because of the difficult mathematics behind. Analytical solutions cannot be found in most of the cases, even a proof of existence is often impossible. Now they can simply sketch the solutions.

On the other hand much more complex problems can be solved with the quasi Newton algorithm applied to iterated networks. The shared weight methods were able to solve at most 4 to 5 iterations. Networks with more hidden layers presented an impassable bottleneck for backpropagation. With the direct weight update by the quasi Newton algorithm we were able to compute the twentieth root of several functions without problems.

Acknowledgements

This research was sponsored in part by the German Federal Ministry of Education, Science, Research and Technology under grant number 01 IN 505 B.

References

[1] Kuczama M., Choczewski B., Ger R., *Iterative Functional Equations*. Cambridge University Press, Cambridge, 1990.

[2] Feigenbaum M. J., *Universal Behaviour in Nonlinear Systems*. Los Alamos Science, Los Alamos, 1978

[3] Martinetz T., Protzel P., Gramckow O., Sörgel G., *Neural network control for steel rolling mills*. In Kappen B., Giele S., *Neural Networks: Artificial intelligence and industrial application*. Springer, Berlin, 1995.

[4] Waibel A., *Modular construction of time-delay neural networks for speech recognition*. Neural Computation 1:39-46, 1989.

[5] Kindermann L. *Computing Iterative Roots with Neural Networks.* Proceedings of the Fifth Conference on Neural Information Processing, ICONIP'98 Vol. 2:713-715, 1998

[6] Rumelhart D. E., McClelland J. L., *Parallel Distributed Processing*. MIT-Press, Cambridge, 1984.

[7] Bishop C.M., *Neural Networks for Pattern Recognition.* Clarendon Press, Oxford, 1995