Learning to Tell Two Spirals Apart

Kevin J. Lang and Michael J. Witbrock

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Alexis P. Wieland recently proposed a useful benchmark task for neural networks: distinguishing between two intertwined spirals. Although this task is easy to visualize, it is hard for a network to learn due to its extreme nonlinearity. In this report we exhibit a network architecture that facilitates the learning of the spiral task, and then compare the learning speed of several variants of the back-propagation algorithm.

1 The task

In a recent post on the connectionist mailing list, Alexis P. Wieland of the Mitre Corporation proposed an interesting benchmark task for neural networks. The task requires a network with two input units and one output unit to learn a mapping that distinguishes between points on two intertwined spirals. Given the activation of the two input units, which encode the x and y coordinates of a point in the plane, the network should output a one if the point falls on one spiral, and a zero if it falls on the other. The two spirals were specified by means of a C program, which is reproduced in the appendix of this report. The 194 training points generated by this program are shown in Fig 1. The <x,y> points at which the network should output 0's and 1's are represented by black and white dots respectively. The correct response for any input other than one of the training points is unspecified.

Wieland pointed out several features of this task which make it an interesting test of connectionist learning algorithms. Not only does it require the network to learn a highly non-linear separation of the input space, which is difficult for most current algorithms, its 2-dimensional input space makes it easy to plot the network's transfer function in order to study the network's inner workings and development during learning.

This work is supported under National Science Foundation grants EET-8716324 and IST-8520359, and Office of Naval Research contract N00014-86-K-0678.

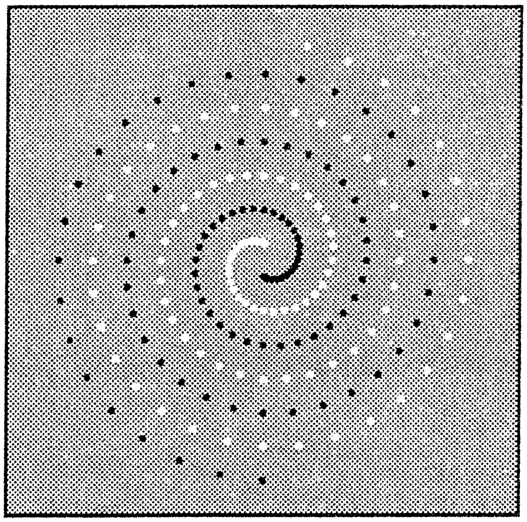

Figure 1: Spiral Training Points

Initially, we attempted to learn the task with "standard" back-propagation nets, containing few layers of hidden units, and connections only between adjacent layers. These experiments failed, convincing us both of the difficulty of the task, and of the need to design a specific network architecture to suit the problem. Our network, which contains 19 units and 138 connections, is reliably able to learn the spiral training set in 20,000 epochs of vanilla back-propagation, and in half that with an enhanced version of the algorithm.

2 The network

Such a highly non-linear problem would clearly benefit from the computational power of many layers. Unfortunately, back-propagation learning generally slows down by an order of magnitude every time a layer is added to a network. This is because the error signal is attenuated each time it flows through a layer, and learning progress

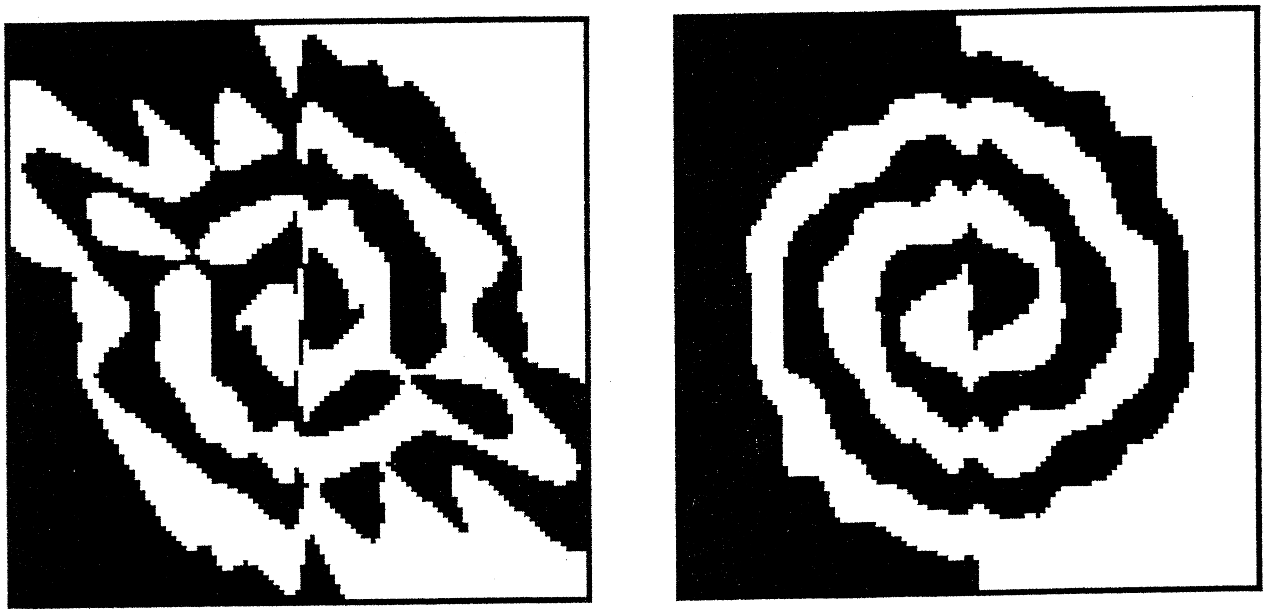

Figure 2: Run A after 19,000 epochs, and Run C after 64,000 epochs on the dense training set

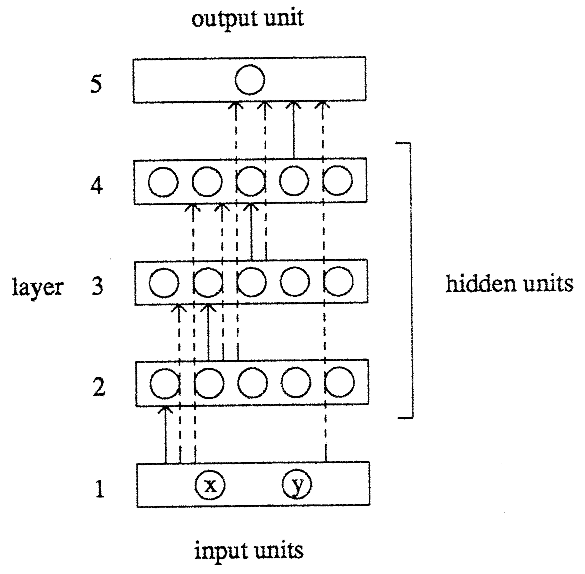

Figure 3: Network Architecture for the Spiral Problem

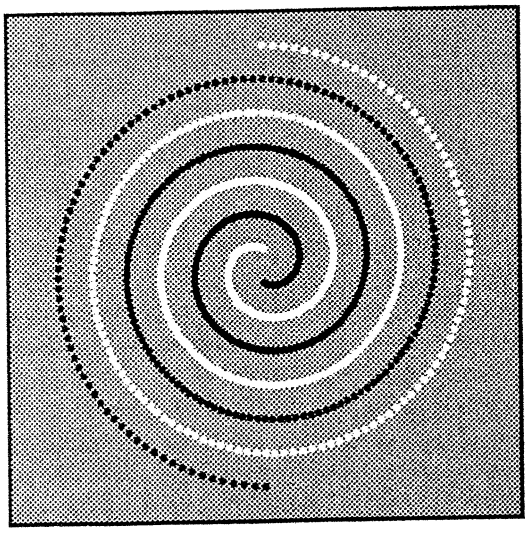

Figure 4: Dense Spiral Training Points

is therefore limited by the slow adaptation of units in the early layers of a multi-layer network. To avoid this problem, we used short-cut connections to provide direct information pathways to all parts of the network. Our connection pattern differs from the usual one in that each layer is connected to every succeeding layer, rather than just to its immediate successor.

Freed from concerns of exponentially slow learning, we were able to use as many layers as we wanted. As a first guess, we tried 5 layers, meaning that the network contains an input layer, 3 hidden layers, and an output layer. The number of units in each hidden layer was then chosen using a crude estimate of the network's information capacity. The task requires the network to know 194 bits of information in order to answer correctly on all of the training cases. Our research group uses the

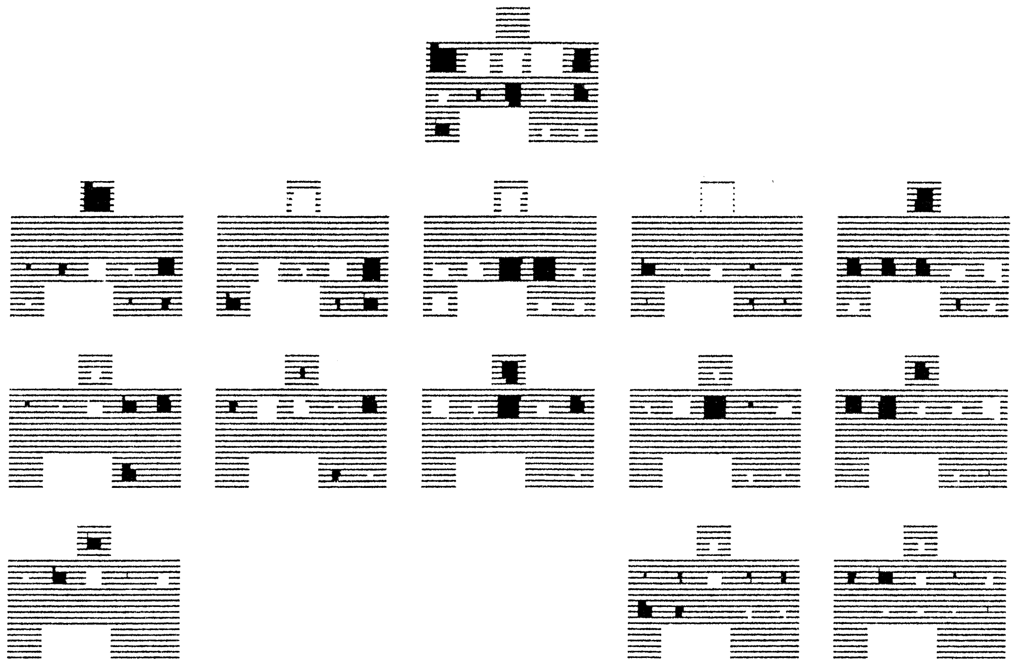
Figure 5: Recursive weight diagram for a four layer network. Each sub-shape in this diagram shows the weights connecting a given unit with all of the other units in the network. The three units on the bottom are the true unit (for implementing biases) and the two input units. The next two layers contain five hidden units each, and the top layer consists of the output unit. White and black blobs represent excitatory and inhibitory weights respectively.

rule of thumb that a connection weight can easily learn 1.5 bits of information, so the network should contain around $194/1.5 = 130$ weights. We built the network with 5 units per hidden layer, resulting in 138 weights. This network is shown schematically in figure 3.

Using vanilla back-propagation, the network was trained on the task 3 times, starting from 3 sets of weights randomly initialized in the inverval $(-0.1, +0.1)$. The parameters started out at $\{\epsilon = 0.001, \mu = 0.5\}$, where ϵ is the gradient scaling parameter, and μ is the momentum parameter. The parameters were increased gradually over 1,000 epochs to final values of $\epsilon = 0.002, \mu = 0.95$. Learning was considered complete when the output unit activation was within 0.4 of the target value of 0 or 1 for every case. The learning times of the three runs, which we designated A, B and C, were 18,900, 22,300 and 19,000 epochs respectively.

The left hand plot in figure 2 shows how the network with weights learned by run A responds to input points in the square region surrounding the training set. White and black dots indicate whether the network's output unit is more or less active than 0.5. Although the network's response pattern is not a perfect spiral, it does fit all of the training points. The network hasn't run out of capacity, it is just underconstrained. It is interesting to observe that the "spiralness" of the response pattern is best near the center of the picture, where the training points are the densest (compare figure 1). To demonstrate that the network could learn a denser training set and generate a nicer picture, we trained the network on the denser training set pictured in figure 4. This set, which contains 4 times as many points, took 64,000 epochs to learn, and resulted in a network whose response pattern is shown in the right hand plot of figure 2.

To find out which aspects of our network design were really necessary, we attempted to train simpler networks that contained approximately the same number of weights. For example, a 4-layer network with 10 units per hidden layer and without short-cut links between non-adjacent layers contained 151 weights, but was unable to learn any of the training cases before reaching a state at 14,000 epochs from which it was unable to reduce the error. On the other hand, 4-layer networks *with* short-cut links were able to learn the task, albeit more slowly and less reliably than the 5-layer network that is the focus of this report.

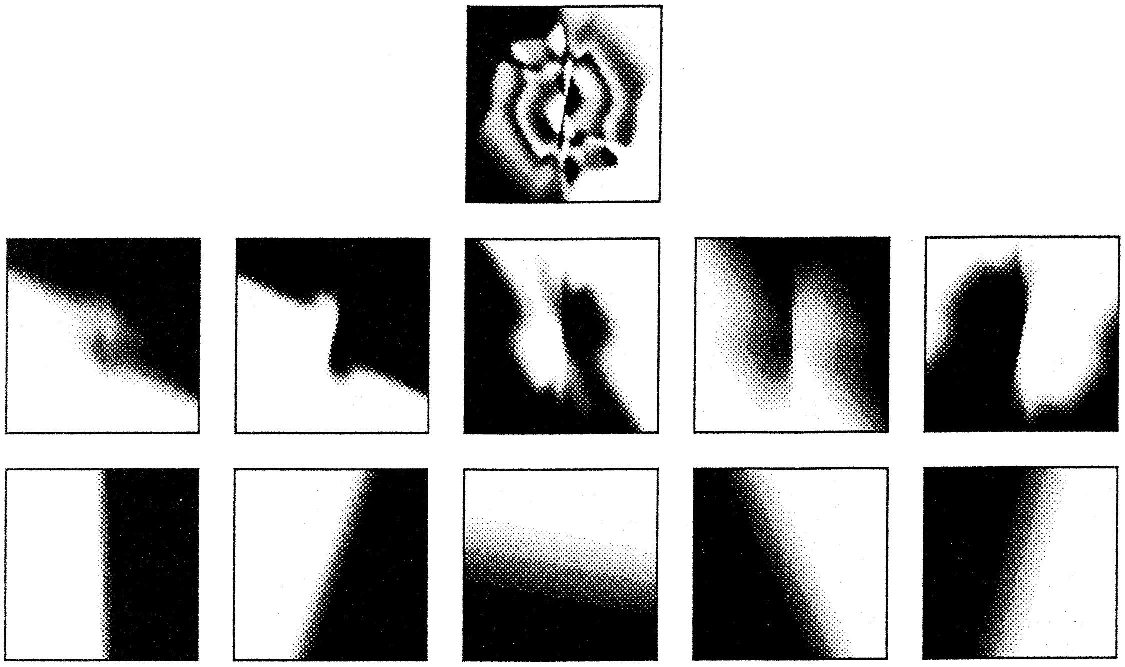

Figure 6: Response function plots for the units in a four layer network. Each plot shows the activation pattern of a single unit as the <x,y> input to the network ranges over a square region containing the spirals. The top plot is for the output unit, and the plots below are for the five units in each of the two hidden layers.

3 The Inner Workings of a Network

Like connectionists everywhere, we wanted to find out how our network works. We were especially curious about the role played by the short-cut connections. The recursive weight display of figure 5 shows that the network did in fact make use of these connections, developing significant weights between all of the layers. [1]

Having verified that all of the network's connections were being utilized, we could learn little more from studying figure 5. Weight displays from real-valued tasks are generally difficult to interpret. Figure 6 provides a clearer picture of the computation performed by the network. Each square in the diagram shows the states assumed by a single unit as the <x,y> input ranges over the region surrounding the training set. Black represents an activation of 0.0, white represents 1.0, and shades of grey represent intermediate values. As one would expect, the units in the first hidden layer perform linear separations of the space at various angles. The units in the second hidden layer form round patterns of various diameters that are used to form the successive turns of the spirals.

[1]To simplify the diagrams in this section, we exhibit a network with only 4, rather than 5 layers. Training this 68-weight network took much longer than the others: 60,000 vs 8,000 epochs of Quickprop.

After studying figures 5 and 6, we concluded that the connection structure of our network facilitates the efficient utilization of its computational resources. The difficulty of this task lies in building up a sufficiently complex response function for the output unit. The job of the hidden units is to provide an appropriate set of basis functions for the output unit to combine into a function describing a spiral. With our fully connected architecture, each unit has a richer selection of possible component functions available than it would have if the unit were only connected to the units in the immediately preceding layer.

4 Two Learning Histories

Figure 8 is a plot of errors versus time for two learning runs of our network on the spiral problem. Although the two runs, A and C, consumed approximately the same number of epochs, the two paths toward a solution were qualitatively different. During run A, the number of errors plunged rapidly early in training as the network deciphered the spiral structure, but ultimately the network had trouble learning the last few cases. During run C, on the other hand, the network took a long time to make a significant dent in the problem, but was then able to clean up all of the cases in short order.

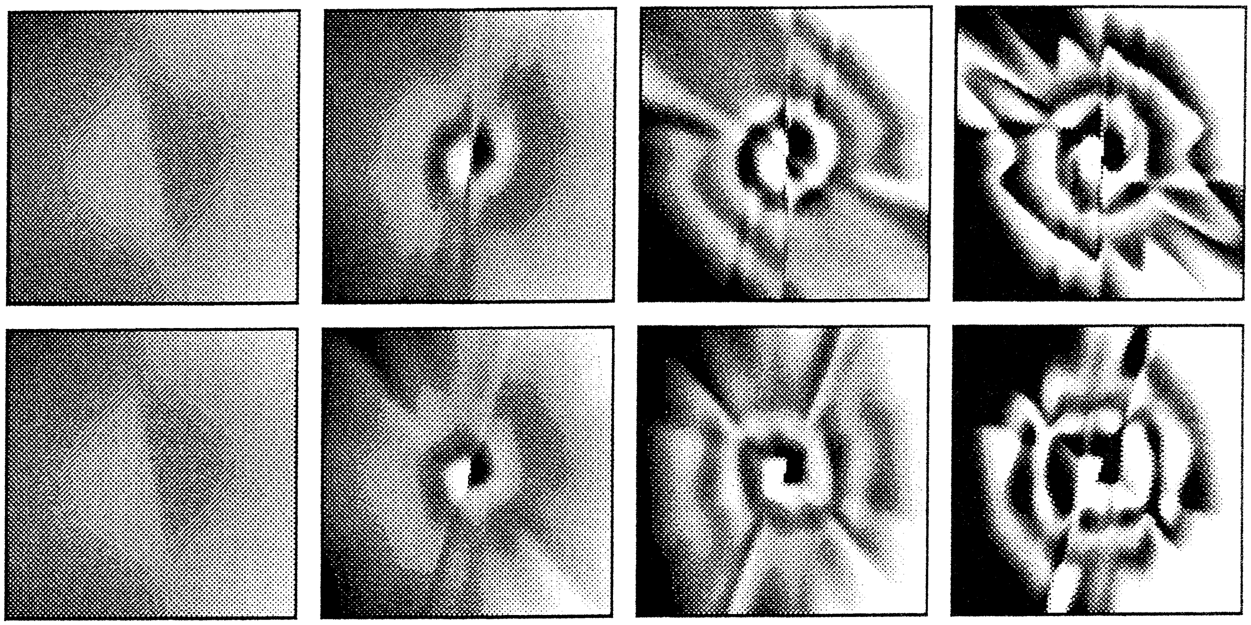
Figure 7: Run A and Run C at four stages during learning

Figure 7 shows the function computed by the network at four milestones in its development during these two training runs. The plots were made at the points in training when the network had learned 25, 100, 150, and all 194 of the cases. The top four pictures are of run A after 4, 6, 8 and 19 thousand epochs, and the bottom four are of run C after 6, 13, 16 and 19 thousand epochs. During both runs, the network first gave the space a weak black-to-white gradation from left to right, and then progressed to the leaf-like pattern shown in the leftmost pictures. The clear spiral shape learned at the next milestone appears in the center because the training set is densest there. At the third milestone, the runs have begun to look visibly different. Run A has generated a nice round central spiral at this point, but has failed to develop much structure in the upper left and lower right quadrants[2].

The network's resources were better distributed among the various cases in run C, in which it was able to develop all of the turns of the spiral in a natural way, whereas in run A it had to create a separate flame-like pattern to account for the outermost cases. A plot of the hidden unit activation patterns after run A showed that a single hidden unit in the third hidden layer is solely responsible for generating this pattern.

Although this attempt to explain network behavior based on inspection of response function plots is admittedly subjective, there are objective reasons for preferring the weights learned during run C. The amount of generalization which the network is capable of can be measured by testing the network on the dense spiral of figure 4 (training was performed on the normal spiral of figure 1). Using the weights developed during run A, the network makes 76 errors, whereas with those developed during run C the network can account for all but 56 of the 770 cases.

Furthermore, the learning trajectory of run C was more susceptible to improvement by a well-known enhancement of back-propagation. Back propagation is often faster when gradient descent is performed with respect to the cross-entropy between actual and desired outputs rather than with respect to the squared distance between the two. The cross-entropy function (described in an appendix) generates strong error signals when the output is completely wrong (1 instead of 0, or 0 instead of 1), and hence is more strongly driven during the initial stages of learning. The course of learning for two runs using cross-entropy is plotted in figure 9. Starting from the initial random weights of run A, cross-entropy back-propagation overcommits even more seriously than ordinary back-propagation to the early, central structure, and takes an extremely long time to learn the last few

[2]Observe that the network's response pattern is skew-symmetric. The degree of symmetry exhibited by the network's solutions is surprising; although the training set is skew-symmetric, the network starts from randomized initial weights that are indifferent to that property.

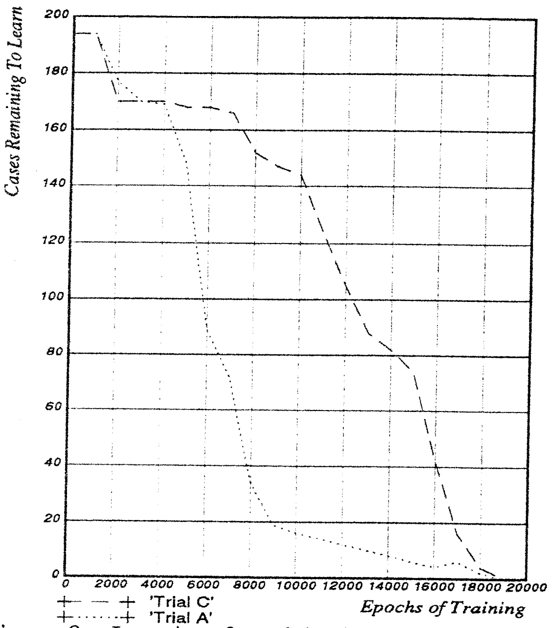

Figure 8: Learning for trials A and C for "vanilla" back-propagation

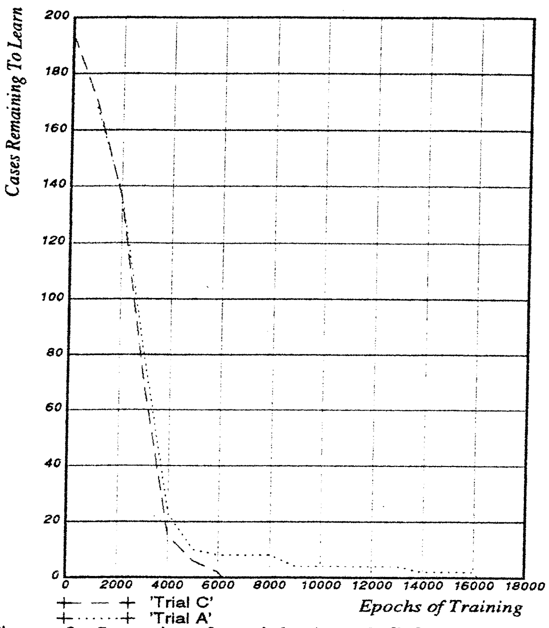

Figure 9: Learning for trials A and C for cross-entropy back-propagation

cases. By comparison, when learning is started from the initial random weights of run C, it proceeds rapidly, producing correct outputs for all cases in less than half the time taken by vanilla back-prop.

5 Comparison of learning algorithms

In order to thoroughly explore the difficulty of the spiral task, we learned it using three variations on back-prop: ordinary error back-propagation with momentum ("vanilla" back-prop), vanilla back-prop with a cross-entropy error function, and "Quickprop". Vanilla back-prop has been described in detail elsewhere [Rumelhart 1986]; cross-entropy back-prop is nearly identical, merely substituting the function described in Appendix A for the usual $(target - output)^2$ error measure. Quickprop is a new learning algorithm, developed by Scott Fahlman [Fahlman 1988], which has shown promise in delivering shorter learning times than the usual versions of back-prop[3].

Instead of performing simple gradient descent, Quickprop uses successive values of the gradient of the error surface in weight space to estimate the location of a minimum. It then changes the weights to move directly towards this minimum. The two principal assumptions of the algorithm are that the error surface is concave, and that the surface is locally quadratic. From these assumptions, it is easy to derive the weight update rule:

$$\Delta w_t = \frac{\partial E/\partial w_t}{\partial E/\partial w_{t-1} - \partial E/\partial w_t} \Delta w_{t-1}$$

While early work with Quickprop showed it to be exceptionally good at learning binary auto-encoders, it was not known whether these gains could be transferred to problems involving continuous valued inputs or outputs. The recent completion of a Quickprop simulator running on a Convex mini-supercomputer, and Wieland's publication of this exemplarily continuous problem gave us an opportunity to test this hypothesis[4]. Table 1 shows that Quickprop did somewhat better than either variation of ordinary back-prop.

Figure 10 shows the progression of learning by Quickprop for the cases discussed in detail above for back-prop[5]. In each case, the initial random weights are the same ones from which back-prop learned, but the course of learning is quite different. Quickprop's weight change algorithm differs significantly from that of back-prop, and

[3]Readers interested in experimenting with Quickprop should read Fahlman's *"Faster-Learning Variations on Back-Propagation: An Empirical Study"* contained elsewhere in this volume. Quickprop requires that some minor modifications be made to the algorithm outlined herein in order to work correctly.

[4]Appendix B gives the parameters which were used with Quickprop for all the trials with 3 hidden layers.

[5]The omission of trial B is simply for consistency. In this trial, Quickprop learned most cases rapidly, spending the last 6200 epochs, more than half the learning time, to learn the last 10 cases.

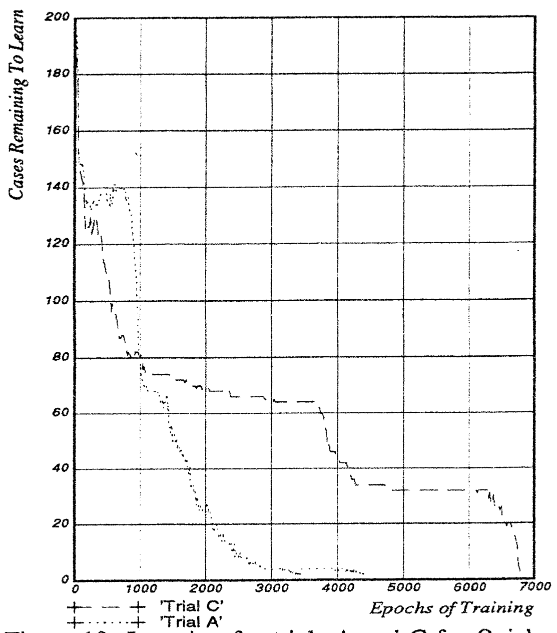

Figure 10: Learning for trials A and C for Quickprop

	Learning Algorithm		
Run	Vanilla BP	Cross Entropy BP	Quickprop
A	18 900	16 200	4 500
B	22 300	8 600	12 300
C	19 000	7 600	6 800
Mean	20 000	10 000	8 000

Table 1: Comparison of Error Measures

consequently the path it follows over the error surface in weight space may also differ significantly. In both trials, Quickprop learned many of the cases very rapidly, accounting for a quarter of them within the first 100 epochs. From there, in trial A Quickprop expended many epochs without learning any new cases (and, in fact, "forgetting" some that it had learned), before rapidly acquiring the rest of the task. In Trial C, it continued to learn rapidly until it had more than half the cases correct, but then became "stuck", eventually taking half again as long to solve the problem as it had in Trial A.

6 Conclusions

In the past, there has been widespread reluctance to use networks with large numbers of hidden layers. This reluctance has been based on the fact that the back-propagated error signal suffers considerable attenuation as it passes through each successive layer of the network [Plaut 1986]. However, learning of highly non-linear problems, such as the one discussed in this paper, can be greatly facilitated by multiple hidden layers. By providing "short-cut connections", which connect non-adjacent layers, the worst effects of attenuation can be avoided, allowing the network to use the extra flexibility that multiple hidden layers afford.

Using such "short-cut connections" it is relatively easy for any of the three variations on back-propagation tested in this paper to learn the highly non-linear spiral problem. The fastest times for learning this problem were obtained using the recently developed Quickprop procedure. The success of this new procedure on this highly non-linear continuous task suggests that attempts to apply it to larger, real-world tasks are warranted.

Acknowledgements

The authors would like to thank Alexis Wieland for proposing an interesting learning problem, Richard Szeliski for assistance with graphics, Scott Fahlman for help with the Quickprop simulator, Michael Franzini for programming work on the Convex connectionist simulator, and David Touretzky for his prompt and insightful comments on drafts of this paper.

References

[Fahlman 1988] Fahlman, S.E. *An Empirical Study of Learning Speed in Back-Propagation Networks*, Tech. Report CMU-CS-88-162, Carnegie Mellon University, June, 1988.

[Plaut 1986] Plaut, D.C., Nowlan, S.J., and Hinton, G.E. *Experiments on learning by back propagation*, Tech. report, Carnegie Mellon University, June, 1986.

[Rumelhart 1986] Rumelhart, D.E., McClelland, J.L., and the PDP Research Group. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Vol I. 1986. The MIT Press, Cambridge.

Appendices:

A Cross-entropy Error Measure

$$C = \sum_{j,c} d_{j,c} \log_2(y_{j,c}) + (1 - d_{j,c}) \log_2(1 - y_{j,c})$$

where $d_{j,c}$ is the desired activation of output unit j in case c and $y_{j,c}$ is its actual activation.

B Parameters used during Quickprop Learning

A single set of parameters was used for the entire duration of all the runs of Quickprop.

ϵ	=	0.002
μ	=	1.5
weight decay	=	0.001
error measure	=	*hyperbolic arctangent*
sigmoid prime offset	=	0.0

C Program used to generate spiral training set

```
/*****************************
 **
 **   mkspiral.c: A program
 **   to generate training
 **   data for a network with
 **   2 inputs and 1 output.
 **
 **   Any questions or comment
 **   on this task contact:
 **      Alexis P. Wieland
 **      MITRE Corporation
 **      (703) 883-7476
 **      wieland@mitre.ARPA
 *****************************/

#include <stdio.h>
#include <math.h>

main()
{
 int i;
 double x, y, angle, radius;

 /* write spiral of data */
for (i=0; i<=96; i++) {
  angle = i * M_PI / 16.0;
  radius = 6.5 * (104-i)/104.0;
  x = radius * sin(angle);
  y = radius * cos(angle);
  printf("%8.5f %8.5f %3.1f\n",
      x,  y, 1.0);
  printf("%8.5f %8.5f %3.1f\n",
     -x, -y, 0.0);
 }
}
```