2. Neural Networks: Basic Concepts.

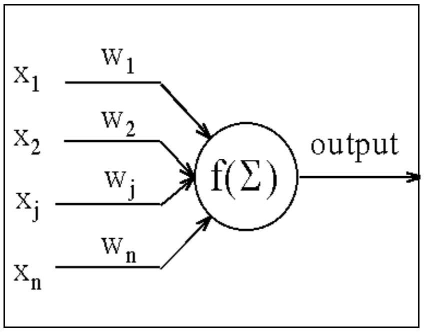

Figure 1: sketch of a neuron.

The basic element of a neural network (NN) is a neuron, also referred to as a node (see figure 1). Each node receives input signals via a series of n connections, which we can label $x_1..x_n$. Each connection has a weight associated with it, which we can label $w_1..w_n$. The total weighted input seen by a neuron is the sum of all x_iw_i. That is, each input signal is multiplied by the weight of the connection it is transmitted on, and then the products of all these multiplications are added together.

For example, in figure 2 we see a neuron with inputs x_1=4, x_2=.2, x_3=-7.3, and x_4=-.35. Associated with these inputs we have connections of w_1=1.2, w_2=.-3.7, w_3=-2.1, and w_4=2.5. This being the case, the neuron receives a total input of 4*(1.2) + .2*(-3.7) + (-7.3) *(-2.1) + (-3.5)*(2.5) = 10.64. This weighted input is then fed into the neuron's transfer function.

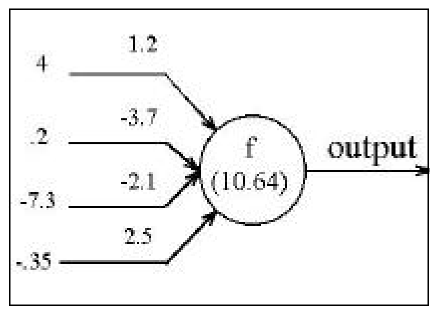

Figure 2: a neuron with particular values for inputs & weights.

Various transfers functions are available to someone designing a NN, some of which are presented in figure 3. Figure 3(A) shows a step transfer function, with threshold of 1. Figure 3(B) shows another step transfer function, this time with a threshold of 2. Figure 3© shows a ramp transfer function, with maximum value of .75. Finally, figure 3(D) shows a sigmoidal transfer function, which has an output of 0 for inputs smaller than 1, starts increasing when its input reaches 1, and saturates to an output value of 1 when its input reaches 2. With these and any other transfer function, the output of the transfer

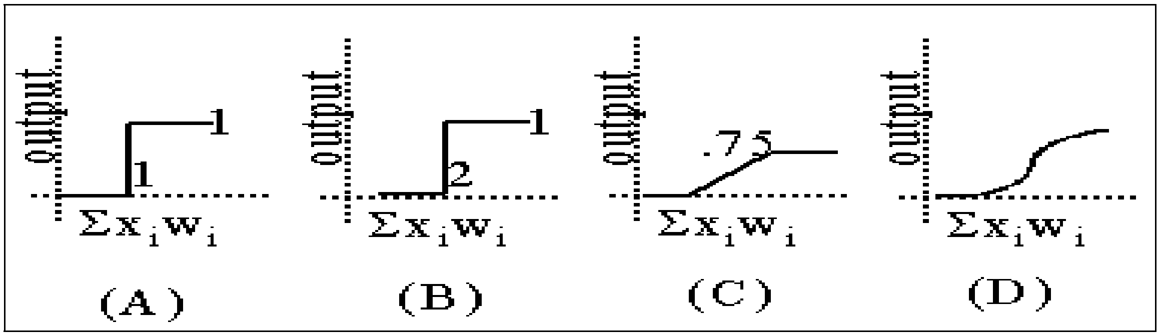

Figure 3: examples of transfer functions.

function is placed on the neuron's output connection.

Even though the two neurons I have used as examples here both have 4 inputs, neurons in general can have any number of inputs.

As early as 1966, Papert and Minsky (1966) proved that a single neuron can solve very few problems. In particular, they showed that a single layer network can learn to produce correct outputs only when the input combinations are linearly separable. For example, take the case of the binary

x	y	and(x,y)
0	0	0
0	1	0
1	0	0
1	1	1

Figure 4: truth table for binary function and().

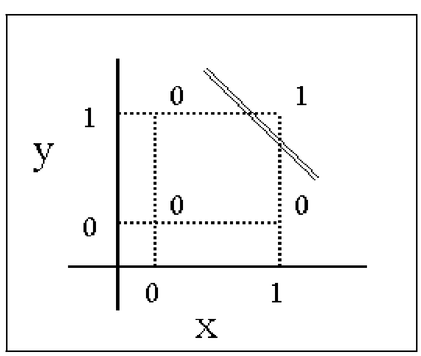

Figure 5: graphic representation for and() function.

function and(), which has the truth table shown in figure 4. We can also represent this function with a two-dimensional graph, each input variable being displayed on a different axis (in general, a function with n inputs could be represented in this way using an n-dimensional graph). Figure 5 represents the and() function using this method. In addition, figure 5 shows that we can find a line that divides the input space where the function produces an output of 1 from the input space where the function produces an output of 0. (In general, for a function with n inputs, we would need to find an (n-1)dimensional plane dividing the input space.) The fact that this line exists means that a neuron can divide, or recognize, the input combinations that need to produce a 1 from the input combinations that need to produce a 0. For this simple case, we can use a neuron with weights as indicated in figure 6, and a step transfer function with threshold of 2.

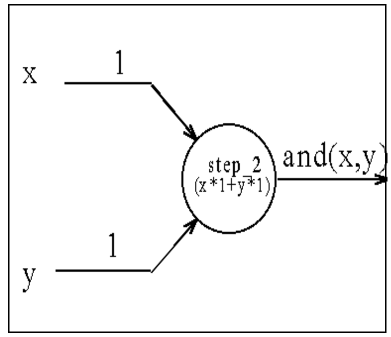

Figure 6: neuron that generates outputs identical to and() function.

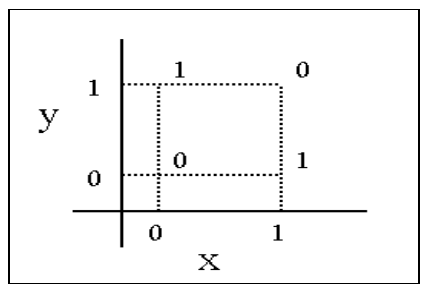

Figure 7: graphic representation of binary function xor().

What Papert and Minsky showed was that there were very simple functions that a single layer network cannot imitate. For example, take the case of the binary function xor(). The xor() function produces an output of 1 when exactly one of its two input lines has a value of 1. No neuron can duplicate the output of an xor() binary function, shown graphically in figure 7, since we cannot find a single line that can divide the input space where the function produces an output of 1 from

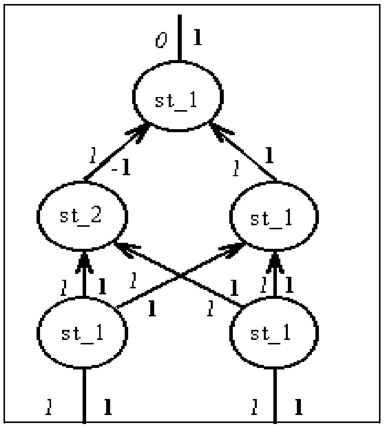

Figure 8: xor NN correctly processing input (1,1).

the input space where the function produces an output of 0. This was a very influential conclusion, since xor() is considered a fairly simple function. If a neuron could not solve this problem, most functions would suffer the same fate and be insoluble.

A solution to this problem is found once we connect several neurons together, forming a multi-layer network. Take, for example, the network shown in figure 8. From a "black box" perspective, this network looks exactly like a single neuron trying to solve the xor problem; it has two inputs and one output. Internally, though, it has several neurons working together to solve the xor problem, which they manage to do. Figure 8 illustrates the network while processing input (1,1). The bold numbers to the right of a connection represent the connection's weight, and the number in italics to the left of the connection represents the signal present on that connection. St_1 represents a step transfer function with threshold of 1, while st_2 represents a step transfer function with threshold of 2. This network produces a correct output for all four possible input combinations.

Therefore, by combining neurons to form a neural network, we have managed to come up with a computation device more powerful than any one neuron. In fact, Siegelmann and Sontag (1992) have shown that NN are Turing powerful. That is, anything that can be computed by a digital computer can be computed by a correctly configured NN.

Although, as we have just seen, NN can compute a large class of functions, their real power comes from the fact that, unlike other computing devices, no explicit description of their behavior needs to be provided. Rather, NN can learn to approximate a function by being presented with input-output pairs. The network is presented with an input, and it propagates signals among its connections according to the weights and transfer functions it has at the moment. Eventually we get a signal on the network's output connections. If this output is the same as the desired output for the input just presented, then nothing is modified. On the other hand, if the actual and the desired outputs are not the same, then the network's weights are modified in such a way as to minimize the network's error. This process is repeated for a predetermined number of epochs, or until a predetermined error is achieved.

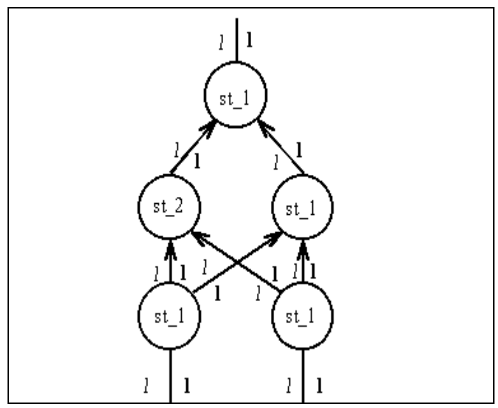

Figure 9: NN that fails to solve the xor problem.

For example, lets say we want to, once again, have a NN learn the behavior of the xor function. If the network we have is the one shown in figure 9, then the correct output will be produced for three of the four possible inputs. For input (1,1), though, the network will produce an incorrect output of 1. Notice that this network differs from the one presented in figure 8 only by the value of one of its weights (the weight with a value of -1 in figure 8 has a value of 1 in figure 9). Although we could try to "fix this problem" by tinkering with the NN weights until we find a combination that responds correctly to all input possibilities, this would be impractical when dealing with large networks, and/or with larger input combinations. What we need is an automated process by which the network’s weights can be modified so as to decrease the error. One such method was developed by Rumelhart, Hinton, and Williams (1986). The method is called the generalized delta rule, but is commonly known as standard backpropagation, since it operates on the principle of letting the network compute an output, calculating an error by comparing this output with the desired output, modifying the weights directly connected to output nodes, and then communicating the error for each output node back towards the input nodes, eventually computing weight modifications for all nodes of the network.

The learning method outlined above can be used for any network that does not contain feedback loops. It cannot be used with networks where propagating an error signal from the output nodes back to the input nodes would create an infinite loop. In the network shown in figure 10, node *a* would propagate its error to node *d*, node *d* to node *c*, node *c* to node *b*, and node *b* back to *a*, creating a process that would never finish. This type of network, though, is needed in order to process time-dependent information. For example, we might be interested in training a NN to recognize when two consecutive digits of a binary stream are the same. If the stream were “1 0 1 0 0", we would like the network to output “0 0 0 0 1". Notice that the NN needs to react differently to the first (or second) 0 than to the third 0 because of what has happened in the input stream previously. The network needs, in effect, to store information about past inputs. Only a network with feedback loops can achieve this type

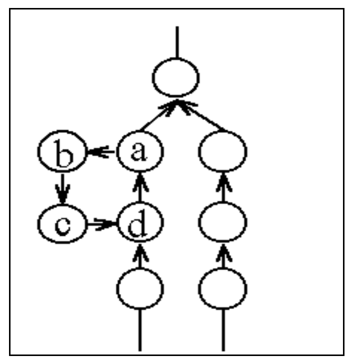

Figure 10: NN with recurrent connections.

of behavior.

In order to train this type of network, called a recurrent neural network (RNN), extensions to the standard backpropagation algorithm such as Recurrent Backpropagation, and Backpropagation Through Time (BPTT) have been devised that take into consideration this type of connection. A good review of these and other learning algorithms has been prepared by Pearlmutter (1990).

Bibliography

Antonisse, J., (1989) A New Interpretation of Schema Notation that Overturns the Binary Encoding Constraint. *Proceedings of the third International Conference on Genetic Algorithms.* San Mateo, California. Morgan Kaufmann, pp. 86-91.

de Garis, H., (1996) CAM-BRAIN: The Evolutionary Engineering of a Billion Neuron Artificial Brain by 2001 Which Grows/Evolves at Electronic Speeds Inside a Cellular Automata Machine (CAM), *Lecture Notes in Computer Science – Towards Evolvable Hardware, Vol. 1062*, Springer Verlag, pp. 76-98,

de Lima, E. (1997) Assigning Grammatical Relations with a Back-off Model. To appear in *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing.* Available for download at http://xxx.uni-augsburg.de/format/cmp-lg/9706001

Dow, R., Sietsma, J. (1991) Creating Artificial Neural Networks That Generalize. *Neural Networks, 4(1),* pp. 67-79.

Elman, J. L., (1991) Distributed Representations, Simple Recurrent Networks, and Grammatical Structure. *Machine* Learning, pp. 71-99

Frasconi, P., Gori, M., and Soda, G., (1992) Local feedback multilayered networks. *Neural Computation, 4(1),* pp. 120-130.

Fullmer, B., Miikkulainen, R., (1991) Using marker-based genetic encoding of neural networks to evolve finite-state behavior. *Proceedings of the first European Conference on Artificial Life.* Paris, pp. 253-262.

Gasser, M., Lee, C., (1991) A Short-Term Memory Architecture for the Learning of Morphophonemic Rules. *Advances in Neural Information Processing Systems 3*. Pp. 605-611.

Goldberg, D., (1989) Sizing Populations for Serial and Parallel Genetic Algorithms. *Proceedings of the Third International Conference on Genetic* Algorithms. Morgan Kaufmann, pp. 70-79.

Hermjakov, U. (1997) Learning Parse and Translation Decisions from Examples with Rich Context. *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics,* pp. 482-489.

Holland, J., (1975), *Adaptation in Natural and Artificial Systems,* Ann Arbor. University of Michigan Press.

Jain, A., (1991) Parsing Complex Sentences with Structured Connectionist Networks. *Neural Computation*, 3, pp. 110-120
Kitano, H., (1994) Designing Neural Networks using Genetic Algorithm with Graph Generation System, *Complex Systems, 4*, pp. 461-476.

Langacker, R. (1985) *Foundations of Cognitive Grammar, Vol. 1: Theoretical Prerequisites*. Stanford University Press.

Lawrence, S., Giles, C., and Sandiway, F. (1998) Natural Language Grammatical Inference with Recurrent Neural Networks. Accepted for publication, IEEE Transactions on Knowledge and Data Engineering.

Miikkulainen, R. (1996) Subsymbolic Case-Role Analysis of Sentences with Embedded Clauses. *Cognitive Science, Jan-Mar, Vol. 20 Number 1*, pp. 43-73.

Munro, P., Cosic, C., and Tabasko, M. (1991) A network for encoding, decoding, and translating locative prepositions. *Connection Science 3*, pp.225-240.

Narenda, K., Parthasarathy, K. Identification and control of dynamical systems using neural networks. *IEEE Transactions on Neural Networks, 1(1)*, pp. 4-27.

Negishi, M. (1994) Grammar Learning by a Self-Organizing Network. *Advances in Neural Information Processing Systems, Vol 5*. MIT Press, pp. 27-34.

Nenov, V., Dyer, M. (1994) Perceptually Grounded Language Learning. *Connection Science, Vol. 6, No. 1*, pp.3-41.

Papert, S., Minsky, M. (1966) Unrecognizable Sets of Numbers. *Journal of the ACM 31, 2, April*, pp. 281-286.

Pearlmutter, B. (1990) Dynamic Recurrent Neural Networks. *Report CMU-CS-90-196*, Carnegie Mellon University.

Romaniuk, S. (1994) Applying crossover operators to automatic neural network construction, *Proceedings of the First IEEE Conference on Evolutionary Computation*, IEEE, New York, NY. pp. 750-752.

Rummelhart, D., Hinton, G.E., and Williams, R. (1986). Learning Internal Representations by Error Propagation. *Parallel and Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1,*. MIT Press, pp.318-362.

Rummelhart, D.E. and McClelland, J.L. (1986) On Learning the Past Tenses of English Verbs. *Parallel Distributed Processing. Explorations in the Microstructure of Cognition:*

Vol. 2, pp. 216-271, Cambridge, MA. MIT Press, pp. 216-271.

Schaffer, J., Caruana, R., Eshelman, L., and Das, R. (1989) A Study if Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization. *Proceedings of the third International Conference on Genetic Algorithms*. San Mateo, California. Morgan Kaufmann, pp. 51-60.

Siegelmann, H. And Sontag, E. (1992) On the Computational Power of Neural Nets. *Proceedings of the Fifth ACM Workshop on Computational Learning Theory*, New York, ACM, pp.440-449.

Skut, W., Brants, T., Krenn, B., et. al. (1998) A Linguistically Interpreted Corpus of German Newspaper Text. *Presented at the ESSLLI-98 Workshop on Recent Advances in Corpus Annotation*. Saarbrucken, Germany.

Smolensky, P., Mozer, M. (1989) Skeletonization: A Technique for Trimming the Fat from a Network via Relevance Assessment. *Advances in Neural Information Processing Systems 1*, Morgan Kaufmann, pp. 107-115.

Solla, S., Le Cun, Y., Denker, J (1990) Optimal Brain Damage. *Advances in Neural Information Processing Systems 2,* Morgan Kaufmann, pp.598-605.

Stolcke, A. (1990) Learning Featured-based Semantics with Simple Recurrent Networks. *TR-90-015*. International Computer Science Institute, Berkeley, CA.

Stork, D., Hassibi, B. (1993) Second order derivatives for network pruning: Optimal Brain Surgeon. *Advances in Neural Information Processing Systems 5*, Morgan Kaufmann, pp.164-171.

Tanese, R. (1989) Distributed Genetic Algorithms. *Proceedings of the third International Conference on Genetic Algorithms*. San Mateo, California. Morgan Kaufmann, pp. 434-439.

Weiss, S., Kulikowski, C. (1991) *Computer Systems that Learn. Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems.* Morgan Kaufmann. San Mateo, California.

Whitley, D., Starkweather, T., and Bogart, C. (1990) Genetic algorithm and neural networks: optimizing connections and connectivity. *Parallel Computing 14*, pp.347-361.

Wilson, W. (1993) A Comparison of Architectural Alternatives for Recurrent Networks. *Proceedings of the Fourth Australian Conference on Neural Networks*, Melbourne, Australia, pp. 9-19.