One-Step Neural Network Inversion with PDF Learning and Emulation

Leemon Baird
Department of Computer Science
US Air Force Academy
Colorado Springs, CO 80840
www.leemon.com

David Smalenberger
Department of Computer Science
US Air Force Academy
Colorado Springs, CO 80840

Shawn Ingkiriwang
Department of Computer Science
US Air Force Academy
Colorado Springs, CO 80840

Abstract — **We present two new types of neural networks (both of which can be trained with ordinary error backpropagation) and we present a new algorithm for learning a probability density function (pdf) from example vectors. It is normally difficult to invert a neural network, but for the new *bijective neural network*, it is efficient to find an input producing any desired output, and such an input is guaranteed to exist and to be unique. Furthermore, it can be used as one component in building a *pdf neural network*, which is a neural network with a nonnegative output, and for which it is guaranteed that the integral of the output is exactly 1.0 (as in a pdf function). Both of these can be used for supervised learning using standard error backpropagation. Finally, the new *pdf learning algorithm* is capable of using those networks to learn a pdf given i.i.d. samples drawn from that pdf, and to then generate new vectors from the learned pdf. This, in turn, allows inversion of a function with non-unique inverses, where each inverse is generated with just a single evaluation of the network.**

I. Background

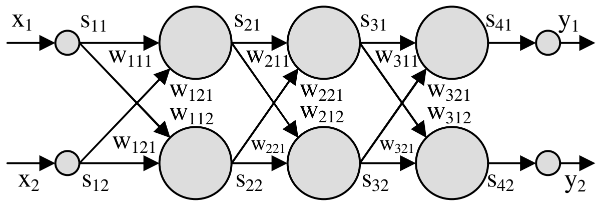
This paper presents a single system, whose components address two unrelated problems that have been widely studied: inverting neural networks, and learning pdfs.

It is a well-studied problem to find an input for a given neural network so that it will give a desired output. For example, to find $\mathbf{x}$ such that $f(\mathbf{x})$ has a given value, where f is a given neural network that has already been trained. There is an extensive literature on this problem (see [1] and [2] for surveys of the field). There are many applications where this would be useful, such as inverse kinematics for robotics [3], or model reference control [4], or optimizing a design where the utility function has been learned by a neural network [5]. Unfortunately, for a typical Multilayer Perceptron (MLP) neural network, this is a difficult problems. Algorithms have been proposed for this problem that perform gradient descent on the input vector [6] [7], or use genetic algorithms [8] [9], or use a number of other approaches. All of these techniques have three problems: finding a single inverse requires many evaluations of the network, the result is only an approximation of the correct answer, and it may not return an answer at all, even if one exists. One obvious solution would be to have the network learn the inverse of the function in the first place, taking $f(\mathbf{x})$ as an input, and giving $\mathbf{x}$ as an output. However, if different $\mathbf{x}$ vectors have the same $f(\mathbf{x})$ value, then an inverse network would end up learning the average of the two rather than learning one or the other.

This can be addressed by both learning and emulating a pdf. If $\mathbf{y}=f(\mathbf{x})$, and the network will be trained with $(\mathbf{x},\mathbf{y})$ pairs, then the network might be trained to learn the probability distribution $pdf(\mathbf{x}|\mathbf{y})$. Merely learning the distribution is insufficient, though. It should be able to efficiently generate new $\mathbf{x}$ vectors according to that distribution given a $\mathbf{y}$ vector. In that case, each x it generates is an inverse value for y, and so the network would be efficiently inverting a non-invertible function. Unfortunately, it is not clear how to both learn the pdf and generate the vectors. If something like a common mixture of Gaussians model is used [10] [11], then it will be difficult to capture complex nonlinearities in a high dimensional space. On the other hand, if a neural network is used to learn the conditional probabilities, it is not obvious how to efficiently generate a new $\mathbf{x}$ vector given $\mathbf{y}$. Such a solution would have further applications beyond just inverting neural networks, such as in recognizing anomalies during data mining. Or a reinforcement learning system might first use real-world data to learn a stochastic model of the MDP to be controlled (which is a pdf of next state given current state and action), and then perform reinforcement learning on that model. The next two sections develop such a system: the *pdf neural network*, whose largest component is the *bijective neural network..*

II. Bijective Neural Network

The *bijective neural network* (BNN) is shown in figure 1. It is a multilayer neural network whose input is n real numbers in the open interval (0,1), and whose output is also n numbers in (0,1). It is similar to a standard multilayer perceptron in that it consists of multiple layers of nodes, each of which takes a weighted sum of its inputs, adds a bias, then applies a nonlinear function to it. The BNN differs from the standard neural network in three ways.



$$s_{1j} = \tanh^{-1}\left(2x_j - 1\right)$$

$$s_{i+1\,j} = \operatorname{arcsinh}\left(c_{ij} + \sinh\left(b_{ij} + \sum_k w_{ikj} s_{ij}\right)\right)$$

$$y_j = \tfrac{1}{2}\tanh\left(s_{4j}\right) + \tfrac{1}{2}$$

Figure 1. Bijective Neural Network (BNN), with inputs **x**, outputs **y**, internal signals **s**, and weights/biases **w**, **b**, and **c**. The number of nodes per layer must equal the number of inputs and outputs.

First, a BNN uses a different nonlinear function in the nodes. Instead of using the traditional nonlinear function:

$$f(x)=\tanh(x)$$

it uses this new one:

$$f(x)=\operatorname{arcsinh}\,(c+\sinh\,(x))$$

where c is an additional weight to be learned. Second, the BNN requires processing on its inputs and outputs. The function y=1/2 tanh(x)+1/2 is applied to every output to squash it down to the interval (0,1), and its inverse is applied to every input. Third, a BNN with n inputs will always have n outputs and n nodes in each layer. This implies that large networks will be deep, with many layers, rather than broad and shallow as is usually the case in traditional neural networks.

Note that the nonlinear function is not numerically stable as written. The inner sinh can generate very large numbers before they are reduced to reasonable size by the arcsinh. However, if the hyperbolic functions are written as exponentials, terms cancel and can be factored to ensure good stability. The function $f(\mathbf{x},c)$ can be calculated safely in C as:

```
if(x>0) {
        t=0.5+c*exp(-x)-exp(-2 * x)/2;
        return x+log(t+sqrt(exp(-2 * x)+t * t));
} else {
        t=-0.5+c * exp(x)+exp(2 * x)/2;
        return x-log(-t + sqrt(exp(2 * x) + t * t));
}
```

The weights between any two layers in the BNN can be thought of as an n by n matrix. As long as all the weight matrices are nonsingular, the BNN is guaranteed to be a bijection from the input hypercube to the output hypercube. This is obvious from the construction of each layer. The inputs are all passed through the inverse hyperbolic tangent. That function is a bijection from its input (which is between 0 and 1, exclusive) and its output (which is between $-\infty$ and ∞). So it is a bijection from the unit hypercube to the entire n-dimensional Euclidean space. The first layer of weights and biases then performs an affine transformation, which is a bijection from the n-space to itself. The nonlinearities in the first layer also perform a bijection from the n-space to itself. This continues for each layer of the network, until the outputs are passed through the hyperbolic tangent functions, which perform the final bijection from the n-space to the unit hypercube. Therefore, the neural network as a whole is guaranteed to be a bijection from the unit hypercube to itself.

This property only holds when the matrices are nonsingular. However, if the initial weights are chosen randomly from small ranges, then with probability 1, the matrices will all be nonsingular. Similarly, after any number of steps of backpropagation with momentum, they will still be nonsingular with probability 1.

Given any set of (nonsingular) weights, the network will implement a nonlinear function that is easy to invert. In fact, the inverse function will be a network identical to the original, but with different weights. It can be found by first swapping the order of the weights. In other words, the weights in the first layer move to the last, the weights in the second layer move to the second-to-last, and so on. Then, each layer must be transformed to implement the inverse function. The weights in a layer can be viewed as a matrix **W** of weights on the connections, and a vector **b** of all the biases:

$$\mathbf{y} = \mathbf{W}\,\mathbf{x} + \mathbf{b}$$

The inverse of that transform is simply:

$$\mathbf{x} = \mathbf{W}^{-1}\,\mathbf{y} + (-\mathbf{W}^{-1}\,\mathbf{b})$$

So the connection weights can be replaced with $\mathbf{W}^{-1}$, and the biases can be replaced with $(-\mathbf{W}^{-1}\,\mathbf{b})$. Finally, each of the nonlinear functions must be replaced by its inverse. It is clear that the inverse of the function:

$$y = \operatorname{arcsinh}\,(c+\sinh\,(x))$$

is the function:

$$x = \operatorname{arcsinh}\,(-c+\sinh\,(y))$$

which means that the nonlinear function can be inverted by simply negating the weight c.

It is interesting that the network is so easy to invert. Given a set of weights, the cost to invert the network is essentially just the cost of doing an n by n matrix inversion for each layer. Once this work is done, the inverse network can be evaluated many times, with each evaluation being done just as efficiently as evaluation of the original network would have been. If the network is very deep, then the original function can be extremely nonlinear and complicated, yet its inverse will always be very efficient to calculate.

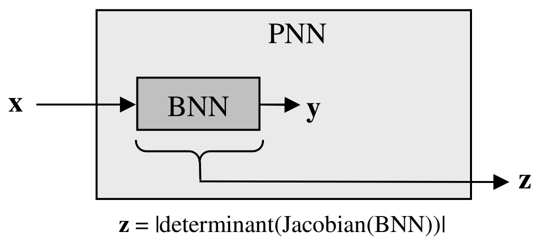


Figure 2. Pdf Neural Network (PNN), with *n* inputs **x**, one output **z**, and the same weights as the BNN inside it. The output of the BNN is discarded. The output of the PNN is the *expansion* of the BNN.

Because the network is made entirely of continuous functions, it can be trained using ordinary backprop with momentum. The only difference between backprop on a traditional network and backprop on a BNN is that the nonlinear functions in the nodes are different, so their derivatives will be different. Therefore the BNN can be implemented by making only a few small changes to existing neural network code. Because of the similarities, the BNN can take advantage of most of the techniques used with traditional networks, such as constraining some weights to remain equal, as in a convolutional network [12].

III. Pdf Neural Network

The pdf neural network (PNN) is shown in figure 2. It is a network with *n* inputs and 1 output. The inputs are each in the interval (0,1), and the output will always be nonnegative. In addition, when the output is integrated over all of input space, the integral is always exactly 1 (assuming, as before, the weight matrices are nonsingular). There may be a number of uses for a network that always has an integral of 1, but the most obvious use is for learning pdf functions. A pdf is nonnegative everywhere, and integrates to 1.

The PNN is formed from a BNN by adding one additional calculation. The input to the PNN is passed to the BNN, which calculates some output (which is ignored). The PNN then calculates the Jacobian across the BNN. The Jacobian is an *n* by *n* matrix formed from the partial derivatives of each output of the BNN with respect to each input. The PNN then outputs the absolute value of the determinant of that matrix.

Call the absolute value of the determinant of the Jacobian of a function its *expansion*. The output of the PNN is simply the expansion of the BNN. This is easily calculated as the product of the expansion of each layer of the BNN. The expansion of a linear transform is the determinant of the weight matrix. The expansion of a layer of nonlinearities is the product of the derivative across each nonlinearity in that layer. These facts make the expansion of a network easy to calculate, and so the PNN is easily built from a BNN.

The PNN can be used to represent a continuous pdf, because its output is always nonnegative and integrates to 1.

```
Pdf learning algorithm

- Loop
    - Generate x randomly with uniform probability
    - Propagate x through the network to get the output z
    - Call backprop to train the network, with a "desired
      output" of 0
    - Set x to a new vector generated by the target pdf to
      be learned
    - Propagate x through the network to get the output z
    - Call backprop to train the network, with a "desired
      output" of z+1
```

Figure 3. The Pdf Learning Algorithm. Both calls to backprop adjust the same weights, and use the same learning rate and momentum. This can work with any neural network, but if a PNN is used to learn the pdf, it can then generate new vectors according to that pdf.

Because all its functions are continuous, it can be trained with ordinary backprop with momentum. All that is necessary is a training set of input vectors **x**, and desired output values pdf(**x**). Once the network has learned the pdf function, it can be given some new **x** vector, and it can calculate the value pdf(**x**). If the pdf(**x**) is low, then **x** is a surprising, anomalous, unusual vector. Detecting this can be useful in areas such as data mining.

However, the trained network can also serve an additional purpose. A random number generator can generate random vectors uniformly in the unit hypercube. If these vectors are passed through the *inverse* of the BNN inside the PNN, then the resulting vectors will be distributed according to that pdf. This is a remarkable result. A traditional neural network can learn a pdf through supervised learning, but then all it can do is evaluate pdf(**x**) for a given **x** vector. It cannot generate new **x** vectors according to that pdf. A traditional network might be trained to give the probability that a given piece of music belongs to a given musical style, but that does not automatically give it the ability to compose new music of that style. A PNN can compose new vectors with whatever pattern was present in the training vectors. Composing a new vector is just as efficient as evaluating the pdf(**x**) for a single x vector.

Note that it is easy to modify the network for conditional distributions *pdf*(**x**|**y**) rather than just *pdf*(**x**). A second neural network can be constructed that is purely a function of **y**, and which is made up of any types of neurons (e.g. sigmoidal or radial basis functions) combined in any number of layers. The output of some of these neurons can then feed in to the sums in the pdf network. During training, the **x** part of a data point is presented to the pdf network, the **y** part is presented to the other network, and then they feed forward and are trained as if they are one network. Once trained, the combined network can be given a **y** vector, and can generate new **x** vectors according to the learned distribution pdf(**x**|**y**).

It will now be shown that the PNN actually does have the properties claimed above. Suppose that a tiny ball is chosen within the input hypercube. If each of the vectors in the ball is passed as input to the BNN, then all the outputs will form a tiny hyperellipsoid. If *e* is the expansion of the BNN at that point, then the volume of the hyperellipsoid will be *e* times the volume of the ball. Suppose now that points are chosen randomly and uniformly in the BNN's output hypercube. If these points are sent through the *inverse* of the BNN, then they will be crowded together *e* times more densely. So the pdf of the new vectors will be *e* times the (uniform) pdf of the random vectors. If the *e* function is trained to be some pdf, then the new vectors will be generated according to that pdf.

IV. The Pdf Learning Algorithm

The previous section showed that a PNN can learn a pdf if sufficient training data is given. Unfortunately, the training data must consist of **x** vectors and their associated pdf(**x**) values. In most practical cases, however, we will simply have a collection of pictures, or recordings, or data vectors generated from some unknown pdf, so supervised learning cannot be used. The *pdf learning algorithm* is shown in figure 3. It is an algorithm that can train a neural network to learn a new pdf, given only a set of i.i.d. vectors generated from that pdf. It does not require that the pdf(**x**) be known for each **x** in the training set. Furthermore, the pdf learning algorithm can be implemented using ordinary backprop code. For each training point, the pdf learning algorithm will call the backprop learning algorithm twice, giving it "desired outputs" chosen in a particular manner.

The operation of this algorithm is intuitive. Imagine learning the function pdf(x) where x is scalar. The target function is a curve. The training set is just a set of real numbers. Every time a training point comes in, backprop is told to make the curve higher by 1 unit for that point. But on every other step it trains the curve to be lower everywhere. The result is that the learned $f(x)$ will tend to be high in regions where the training points occur frequently, and low in regions where they don't. So it will learn to be a pdf.

Of course, this intuition only suggests that the learned curve will be pdf-like, not that it will be the true pdf. There would seem to be variations that are just as promising, such as using a target of $f(\mathbf{x})+2$ instead of $f(\mathbf{x})+1$, or of training with the uniformly-chosen data more often or less often than half the time. However, it turns out that this particular algorithm is required by the mathematics. The $f(\mathbf{x})+1$ must have the +1, and the two types of updates must happen with exactly equal frequencies. This causes it to do stochastic gradient descent on the mean squared difference between the learned function and the true pdf. If it is done on an ordinary neural network, then the network learns the pdf. If it is done on a PNN, then the network learns the pdf, and can also generate new **x** vectors according to that pdf. The next section shows the derivation of this algorithm.

There is one variant of the algorithm that could affect learning speed. Instead of training on uniform just as often as non-uniform, it could be trained with uniform data one third as often, as long as the learning rate was three times larger for the uniform as the non-uniform case. Or, more generally, train $1/n$ as often with a learning rate n times larger. This has the advantage of spending more of the time training on the real data, but the unequal learning rates could cause learning to be even slower. This variant has not yet been explored.

V. Derivation of the Pdf Learning Algorithm

To derive the pdf learning algorithm, it is useful to first look at the derivation of ordinary error backpropagation. In backprop with an infinite training set, the training points **x** are chosen according to some distribution *pdf*(**x**). For each point, the neural network calculates an output $f(\mathbf{x})$, which is compared to the known, desired output $f^*(\mathbf{x})$. The learning algorithm then performs stochastic gradient descent on the total error, which is usually defined as:

$$E = \int \left(f^*(\mathbf{x}) - f(\mathbf{x})\right)^2 pdf(\mathbf{x})d\mathbf{x}$$

The *pdf*(**x**) term represents the fact that backprop will put more emphasis on those training points that appear more often in the training set. If the training set is chosen uniformly, then that term will be a constant 1, and so will disappear.

If the goal is to learn a pdf function, then f^* will be that pdf function. If we are equally interested in getting the function correct at every point in space, then the *pdf*(**x**) term will go to 1 and disappear. Therefore, the pdf learning algorithm should be designed to minimize this total error function:

$$E = \int \left(pdf(\mathbf{x}) - f(\mathbf{x})\right)^2 d\mathbf{x}$$

Multiplying out the quadratic and separating the terms yields:

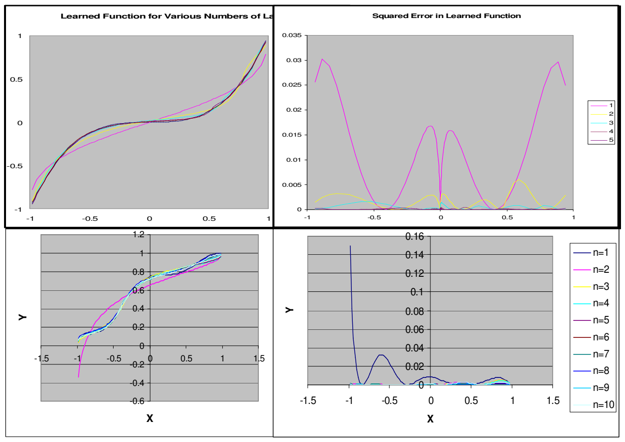


Figure 4. Effect of additional layers on approximation error for the BNN. The learned function (left) and approximation error (right) for different numbers of layers. The error is reduced quickly by the first few layers, and is then almost zero. This is true for a simple cubic function (top) and a more complex 7th order polynomial (bottom).

$$E = \int pdf(\mathbf{x})^2 - 2pdf(\mathbf{x})f(\mathbf{x}) + f(\mathbf{x})^2\, d\mathbf{x}$$

$$= \int pdf(\mathbf{x})^2\, d\mathbf{x} - 2\int pdf(\mathbf{x})f(\mathbf{x})d\mathbf{x} + \int f(\mathbf{x})^2\, d\mathbf{x}$$

Note that the first term is purely a function of the pdf, and is not affected by the weights in the network. Therefore, we could define a new error *E'* with that term deleted, and then doing gradient descent on *E'* would be equivalent to doing gradient descent on *E*. Assume f is a function that is always nonnegative, then simple algebraic manipulations of E' then causes the two remaining terms to look more familiar:

$$E' = \int f(\mathbf{x})^2\, d\mathbf{x} - 2\int pdf(\mathbf{x})f(\mathbf{x})d\mathbf{x}$$

$$= \int (0 - f(\mathbf{x}))^2\, d\mathbf{x} - \int \left(0 - \sqrt{2f(\mathbf{x})}\right)^2 pdf(\mathbf{x})d\mathbf{x}$$

In this final form, each of the two integrals is in the same form as the original backprop error function!

The first integral looks like backprop applied to the neural network whose output is $f(\mathbf{x})$, where $\mathbf{x}$ is chosen uniformly, and where the "desired output" is always 0.

The second integral looks like backprop applied to a neural network whose output is the square root of $2f(\mathbf{x})$, where $\mathbf{x}$ is chosen according to the pdf that is to be learned, and where the "desired output" is always 0. However, in this case there is a minus sign in front of the integral. Gradient descent on this term is equivalent to doing gradient *ascent* with backprop. So backprop will be done with a *negative* learning rate.

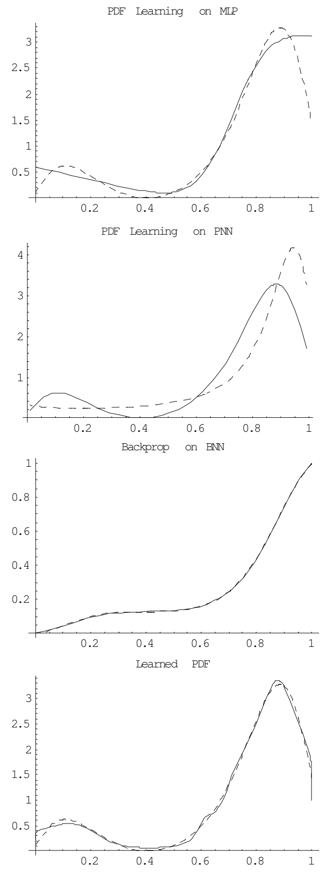


Figure 5. Learned functions (solid) and desired functions (dotted) for both algorithms and all 3 neural networks. The pdf learning algorithm trains a standard neural network to learn a pdf from examples (top). That algorithm also trains a PNN (second). Backprop trains a BNN on a given cdf (third). And the PNN then learns a pdf (bottom).

Both the square root and the negative learning rate look odd, but they can be rewritten in a form that looks much more familiar. In ordinary backprop, an update of a weight *w* would look like this:

$$w \leftarrow w - \alpha\left(f(\mathbf{x}) - f^*(\mathbf{x})\right)\frac{\partial f(\mathbf{x})}{\partial w}$$

If the second term in the *E'* is implemented with backprop, and if we define *g* to be $g(\mathbf{x})$=sqrt(2 $f(\mathbf{x})$), then the weight update becomes:

$$
\begin{aligned}
w &\leftarrow w-(-\alpha)\big(g(\mathbf{x})-0\big)\frac{\partial g(\mathbf{x})}{\partial w} \\
&= w-\alpha\Big(-\sqrt{2f(\mathbf{x})}-0\Big)\frac{1}{2\sqrt{2f(\mathbf{x})}}2\frac{\partial f(\mathbf{x})}{\partial w} \\
&= w-\alpha(-1)\frac{\partial f(\mathbf{x})}{\partial w} \\
&= w-\alpha\big(f(\mathbf{x})-(f(\mathbf{x})+1)\big)\frac{\partial f(\mathbf{x})}{\partial w}
\end{aligned}
$$

This final line looks exactly like the ordinary backprop update for the case where the network output is $f(\mathbf{x})$, and the desired output is $f(\mathbf{x})+1$.

Therefore, the pdf learning algorithm performs gradient descent on E by doing two operations alternately. First, it chooses a random $\mathbf{x}$ uniformly and propagates it through the neural network, then calls backprop to train the output to be 0. Second, it obtains an $\mathbf{x}$ from the training set, which was generated according to the target pdf. It then propagates it through the network to get the output $f(\mathbf{x})$, and calls backprop with a desired output of $f(\mathbf{x})+1$. By repeating these two steps, the algorithm forces backprop to actually do gradient descent on E, and therefore to train the network to learn the pdf of the training set.

VI. Empirical Results

Figure 5 shows the results of testing various combinations of the 5 algorithms and neural networks discussed in this paper. In each figure, the learned function is a solid line, and the target function is dotted. The top figure shows the result of the pdf learning algorithm training a standard neural network using just examples drawn from the distribution (no supervised training). The second shows it training a PNN. The third shows standard backprop training a BNN on a given cdf. The fourth shows the resulting PNN compared to the target pdf. It appears that the algorithms and neural networks described here work with each other, and can also be combined with standard neural networks or algorithms. Figure 4 demonstrates how BNNs gain representational power through more layers (depth) rather than more nodes per layer (breadth).

VII. Conclusion

A new system for learning easily-invertible neural networks has been presented. By combining the pdf learning algorithm with the pdf neural network (which incorporates the bijective neural network), it is possible to learn a highly nonlinear function, then find inverses as efficiently as a single evaluation of the network. Empirical evidence on some simply cases suggests it may have promise, and it may be worth pursuing further with some of the many applications that could be tried.

References

[1] C. A. Jensen, R. D. Reed, R. J. Marks II, M. A. El-Sharkawi, J.-B. Jung, R. T. Miyamoto, G. M. Anderson, and C. J. Eggen, *Inversion of feedforward neural networks: Algorithms and applications*, Proc. IEEE, vol. 87, pp. 1536--1549, Sept. 1999.",

[2] "How to learn an inverse of a function?" comp.ai.neural-networks FAQ. ftp://ftp.sas.com/pub/neural/FAQ7.html#A_inverse

[3] P. Martin and J. R. Millan, "Combining reinforcement learning and differential inverse kinematics for collision-free motion of multilink manipulators," in *Int. Work-Conf. Artificial and Natural Neural Networks, (IWANN'97),* Lanzarote, Spain, June 1997, pp. 1324–1333.

[4] D. A. Hoskins, J. N. Hwang, and J. Vagners, "Iterative inversion of neural networks and its application to adaptive control," *IEEE Trans. Neural Networks,* vol. 3, pp. 292–301, Mar. 1992.

[5] A. P. Weijer, C. B. Lucasius, L. Buydens, and G. Kateman, "Using genetic algorithms for an artificial neural network model inversion," *Chemometrics Intell. Lab. Syst.,* vol. 20, no. 1, pp. 45–55, Aug. 1993.

[6] R. J. Williams. *Inverting a connectionist network mapping by backpropagation of error.* In 8th Annual Conference of the Cognitive Science Society, Hillsdale, NJ, 1986. Lawrence Erlbaum.

[7] A. Linden and J. Kindermann. *Inversion of multilayer nets.* In Proceedings of the First International Joint Conference on Neural Networks, Washington, DC, San Diego, 1989. IEEE.

[8] R. C. Eberhart and R. W. Dobbins, "Designing neural network explanation facilities using genetic algorithms," in *Proc. Int. Joint Conf. Neural Networks*, vol. II, Singapore, 1991, pp. 1758–1763.

[9] R. D. Reed and R. J. Marks, II, "An evolutionary algorithm for function inversion and boundary marking," in *Proc. IEEE Int. Conf. Evolutionary Computation (ICEC'95),* Perth, Western Australia, 1995, pp. 794–797.

[10] Cheeseman, P., Self, M., Kelly, J., Taylor, W., Freeman, D., Stutz, J. . Bayesian classification\ In AAAI 88, The 7th National Conference on Artificial Intelligence, 607--611. AAAI Press. 1988.

[11] Ghahramani, Z. Jordan, M. . Supervised learning from incomplete data via an EM approach\ In Cowan, J., Tesauro, G., Alspector, J., Advances in Neural Information Processing Systems 6. Morgan Kaufmann. 1994.

[12] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Handwritten digit recognition with a back-propagation network. In David Touretzky, editor, *Advances in Neural Information Processing Systems 2 (NIPS*89)*, Denver, CO, 1990. Morgan Kaufman.