Using Neural Networks for Nonlinear Predictive Coding

Kristopher D. Giesing

March 18, 1999

Abstract

Neural networks offer expressive power that is not present in linear systems. However, the relationship between the nonlinear systems of neural networks and their linear cousins is often poorly understood. In this experiment, neural networks were used to implement a nonlinear version of a well understood linear system: linear predictive coding. A neural network was trained to emulate first an FIR filter, then an IIR filter, and finally (in direct correlation with the LPC model) a short segment of a musical sample. In all cases the neural network's performance was a reasonable analogue of the performance of linear systems.

1 Introduction

Since the resurgence in popularity of neural networks and parallel distributed systems in the mid-1980s, neural networks and related systems have been applied to a variety of musical tasks. Many of these have involved learning by example musical rules which, if encoded explicitly, would be cumbersome and in some cases inconsistent with each other. Neural networks have been an attractive solution because of their "black-box" nature; one must simply choose an appropriate sample set to teach the network, and the network itself is expected to derive any features and/or similarities in and among those samples relevant to the tasks

However, by the same token, it is sometimes difficult to ascertain the nature of the calculations that the network is performing to arrive at its conclusions. In some cases one may begin to suspect that the network is performing a simple statistical average over its inputs, providing a merely linear interpolation among its training stimuli as a response to novel input.

The computational power of distributed processing lies in its nonlinear nature. A distributed system with linear transfer functions can be reduced to a simple matrix operation on its input; only systems with nonlinear transfer functions can hope to solve more complex tasks, such as categorization and general rule coding. However, it not always clear when a task provided to a network will exploit the nonlinear nature of the system.

The goal of this experiment was then twofold. First, by applying neural networks to a signal processing task whose linear domain is well understood, we may shed some light on the relationships between nonlinear systems and their linear analogues. Second, it is hoped that the application of such a nonlinear system will provide novel and interesting results for those composers interested in the transformation of sound.

2 Theoretical Background

2.1 Linear Predictive Coding

Linear predictive coding assumes that a signal may be approximated by a linear combination of its previous N samples:

$$x(n) \approx \sum_{k=1}^{N} a_k x(n-k) \tag{1}$$

or

$$x(n) = \sum_{k=1}^{N} a_k x(n-k) + Gu(n) \tag{2}$$

where u(n) is assumed to be a random (stochastic, white-noise) signal. Note that

$$X(z) = \sum_{k=1}^{N} a_k z^{-k} X(z) + GU(z) \tag{3}$$

If we introduce the notion of a transfer function $H(z)$which relates $U(z)$ to $X(z)$, then we have:

$$H(z) \equiv \frac{X(z)}{U(z)} = \frac{G}{1 - \sum_{k=1}^{N} a_k z^{-k}} \tag{4}$$

Note that $H(z)$ is the representation of an infinite impulse response (IIR) filter. Thus the signal is modeled as noise filtered by an IIR filter with poles at the peaks of the signal's spectrum.

2.2 A Neural Network implementation

Neural networks offer a straightforward extension of the LPC model. Here a neural network is tasked with predicting the next sample of a sound based, as above, on its previous N samples. In this case, however, the solution may be a nonlinear combination of those samples.

We shall consider the network to be a feedforward network. Other representations are possible; for example, a recurrent network may be used to give the network a sense of "memory" in addition to the current inputs. However, the feedforward network is arguably the most direct extension of LPC , and offers a simple model for analysis.

Thus we have N inputs (representing the N delayed sample values), feeding into a hidden layer with M units, feeding finally into a single output whose value represents the network's prediction. The equation for such a system is as follows:

$$x(n) \approx F\left(\sum_{i=1}^{N} a_i F(\sum_{k=1}^{M} b_k x(n-k))\right) \tag{5}$$

where $F(x)$ is the nonlinear transfer function of the network (assumed, for simplicity, to be the same function for both hidden and output layers). Since we are here concerned with a continuous rather than discrete output, we use a smooth hyperbolic tangent sigmoid (tansig) transfer function for both hidden and output layers:

$$F(x) \equiv \frac{2}{1 - e^{-2x}} - 1 \tag{6}$$

This function has a slope of 1 near the origin, and curves to approach ± 1 as $x \to \pm\infty$. Thus, the network has near-linear response with small coefficients a_i and b_k. We would then expect the network to be able to find solutions at least as good as their linear relatives.

Once an approximation is found, the network can then be fed a scaled stochastic signal $u(n)$ and will produce its prediction. It is important to note that the network must take into account its own previous predictions, that is, it must operate on the past samples as filtered through the network's nonlinear system. How, then, should the stochastic signal and the prediction be combined? The answer is surprisingly simple; the network's own training error, a sum-squared error over the training set, provides the scale-factor G for the residual $u(n)$:

$$G = \sqrt{SSE/length(set)} \tag{7}$$

where SSE is the sum-squared error produced during training. Thus, G is simply the average error per prediction. Looking at eq. (2) above, we can see that this makes perfect sense in our model.

3 Experimental Setup

The network architecture was chosen to be a three-layer feedforward network with N inputs (representing the N previous sample values) and one output. The number of hidden units was, for simplicity, kept consistent with the number of inputs; that is, for this experiment, $M = N$. The network was constructed and trained using MATLAB's Neural Network Toolbox. The training algorithm used was the Levenberg-Marquardt optimization

technique, which trades memory and longer per-epoch training time to achieve better error minimization. The transfer function for both hidden and output layers was the `tansig` function, as indicated above.

The experiment was divided into three parts based on the content of the training set (corpus) and the generalization technique. These three parts progressed in the complexity of the learning task required of the network, leading up to the full LPC task. In all cases the training set consisted of 1000 samples of a signal, windowed to provide N samples at a time, with the goal of predicting the $N+1$st sample.

In the first experiment, the network was asked to predict the upcoming sample of noise filtered with a finite impulse response (FIR) filter. The training set was the unfiltered noise values, while the desired response was the actual filtered value. Thus, the network was asked to emulate an FIR filter. Because the signal was actually filtered with an FIR filter whose coefficients were known quantities, bounds on the sum-squared error of the linear fit could be calculated directly from the original FIR coefficients, and the network's performance relative to the FIR implementation could be evaluated. The network was asked to generalize by being presented with a new signal; the spectrum of its predictions was then evaluated.

In the second experiment, the FIR task of the first part was generalized to the more germaine infinite impulse response (IIR) task. Here the network was trained to predict the upcoming sample of noise filtered with an IIR filter; the training set in this case was the previous *filtered* noise values, with the target value being again the actual filtered value. Again, the error bound could be calculated directly from the IIR filter parameters, and the network's performance relative to the generating linear system evaluated.

It will be useful in the upcoming discussion to remember the definition of a general infinite impulse response:

$$\begin{aligned} y(n) \quad = a_0 x(n) \quad & + a_1 x(n-1) + a_2 x(n-2) + ... + a_N x(n-N) \\ & + b_1 y(n-1) + b_2 y(n-2) + ... + b_N y(n-N) \end{aligned}$$

Thus, the training set for part 1 consists of the delayed $x(n-k)$ as inputs and $y(n)$ as the target, while the training set for part 2 consists of the delayed $y(n-k)$ as inputs and $y(n)$ as the target.

The third experiment represented the full LPC coding task. Here the network was trained on an actual signal, in this case a fragment of the chorus to Handel's *Messiah*. The error goal in this case could not be calculated directly from the signal, as its characteristics were not analyzed a priori. However, a bound was chosen which seemed to represent the limits of the network's learning. The results were evaluated by simulating the neural network with a random signal and evaluating the spectrum of the result.

4 Results

4.1 FIR Learning Task

In the first part of the experiment, the network was trained to predict 1000 samples of noise filtered with an order 11 FIR filter. The network was presented with the previous 10 samples of the (unfiltered) signal, $x(n-k)$. The other coefficient, a_0, of the FIR filter was the coefficient for the current sample; in terms of eq. (1), this corresponds to the $k=0$ case (which does not appear in the summation); in terms of eq. (2), this corresponds to the scale-factor G. The error bound for the network was then calculated to be

$$E = \sum_{n=11}^{1000} (Gx(n))^2 \tag{8}$$

where $x(n)$ is the input noise signal. Training results are presented in Figure 1. Note that the network learns to within a reasonable approximation of the theoretical bound given by eq. (8). Once the network achieved the desired error goal, it was simulated with two sets of data. In the first set, the network was presented with white noise; its predictions then formed the filtered signal. The fast Fourier transform (FFT) is plotted in Fig. 2, beneath the FFT of the original target sequence (the FIR-filtered signal).As you can see, the network produces a close approximation of the frequency response of our filter. In the second set, the network was asked to filter a completely unrelated signal; this shows that the network is in fact performing a generalized filter task. The peaks of the vocal signal are preserved but scaled in magnitude according to the frequency response of the network. These results are again plotted in Fig. 2in the rightmost quadrants.

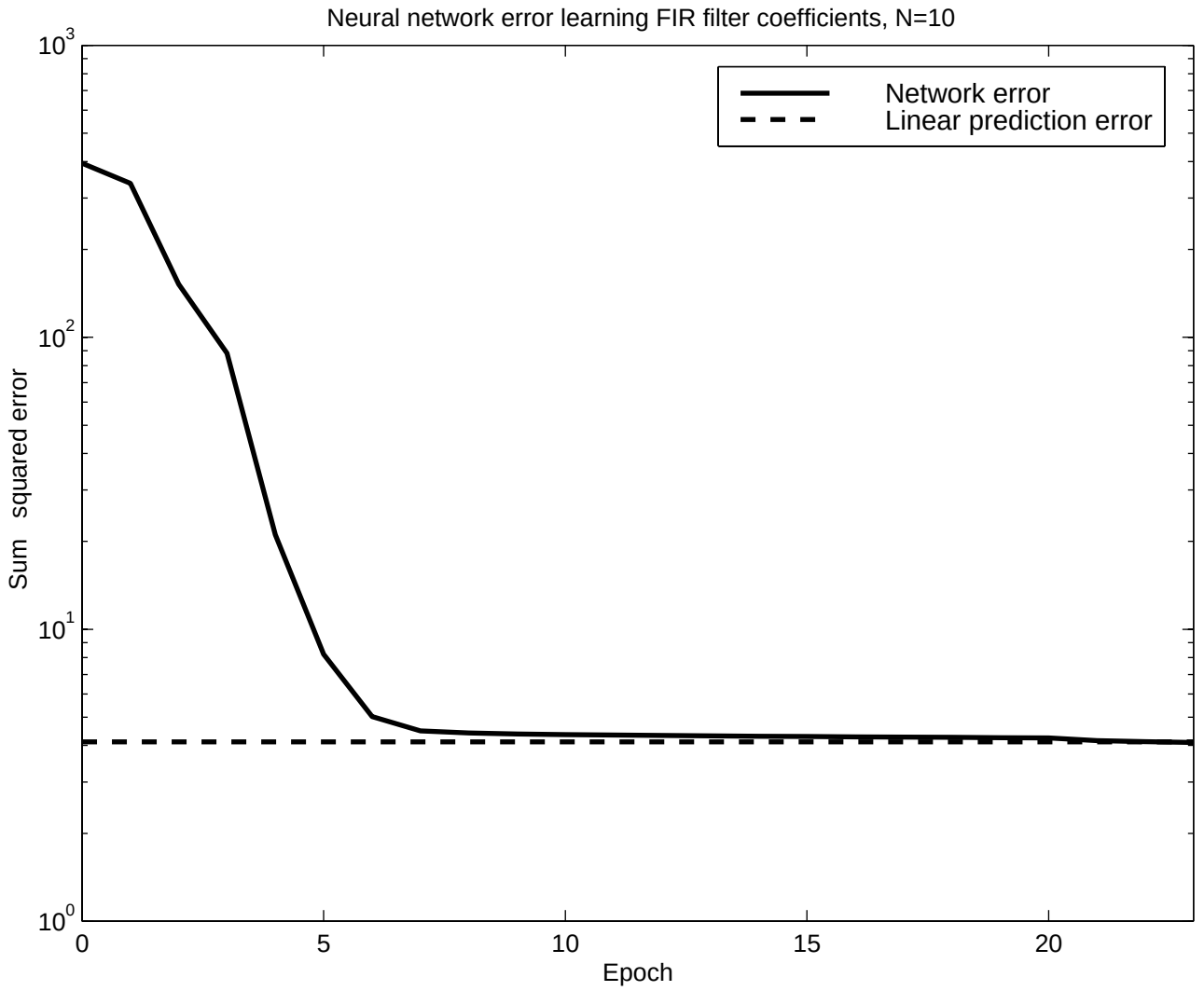

Figure 1: Typical error curve for FIR learning task

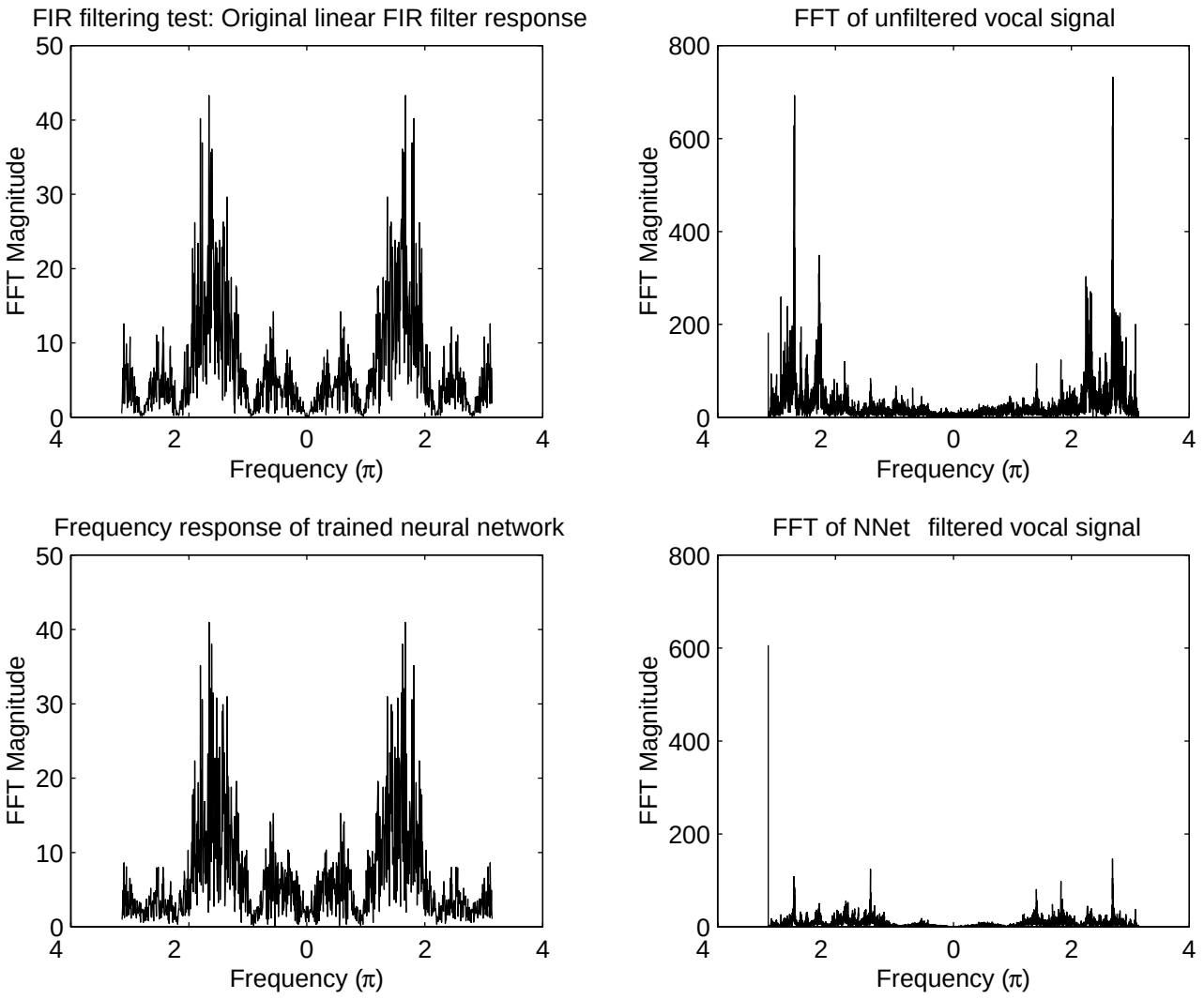

Figure 2: Comparison of various spectra for FIR learning task

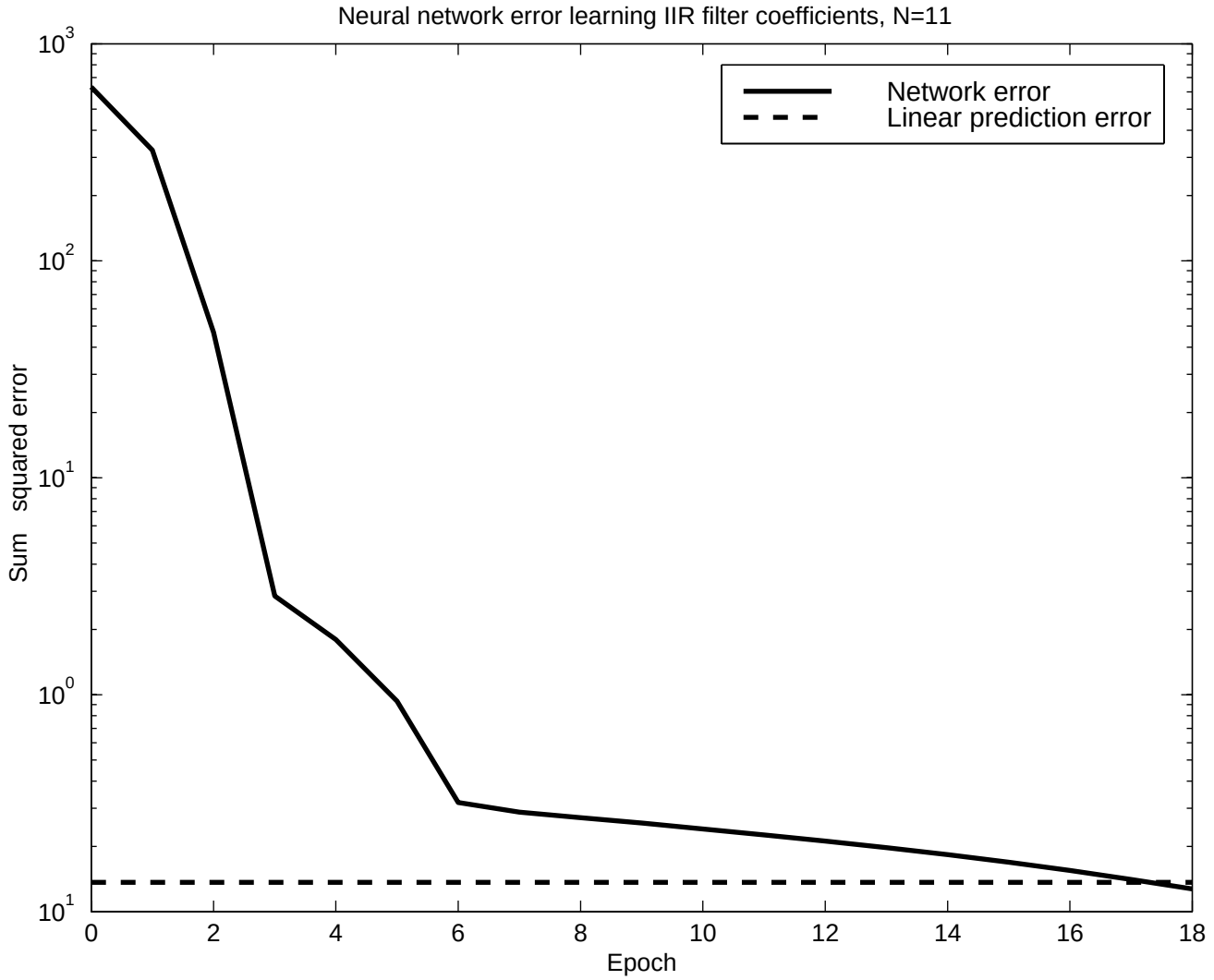

Figure 3: Typical error curve for IIR learning task

4.2 IIR Learning Task

In the second part of the experiment, the network was trained again with 1000 samples. In this case, the network's inputs were the previous filtered noise values $y(n-k)$, while the target was again the filtered value $y(n)$. The IIR filter used was order $N = 22$; the first 11 coefficients were the FIR components (that is, they operated on $x(n-k)$), while the latter 11 coefficients operated on the $y(n-k)$ and represented the coefficients the network was trying to emulate. The error bound was calculated somewhat differently from that in Section 4.1, above; it was the sum of errors calculated for all FIR coefficients:

$$E = \sum_{n=11}^{1000} \left(\sum_{k=0}^{10} a_k x(n)) \right)^2 \tag{9}$$

Because the FIR coefficients for the chosen filter were small, this still provided a reasonably tight error bound. The training error is plotted in Fig. 3.

Again we see that the neural network successfully reached the error goal. The network was then evaluated by simulating its response to the filtered noise signal used in training; the FFT of this response is shown in Fig. 4. However, in presenting the network with a new signal, that is, to generalize to the case where we do not

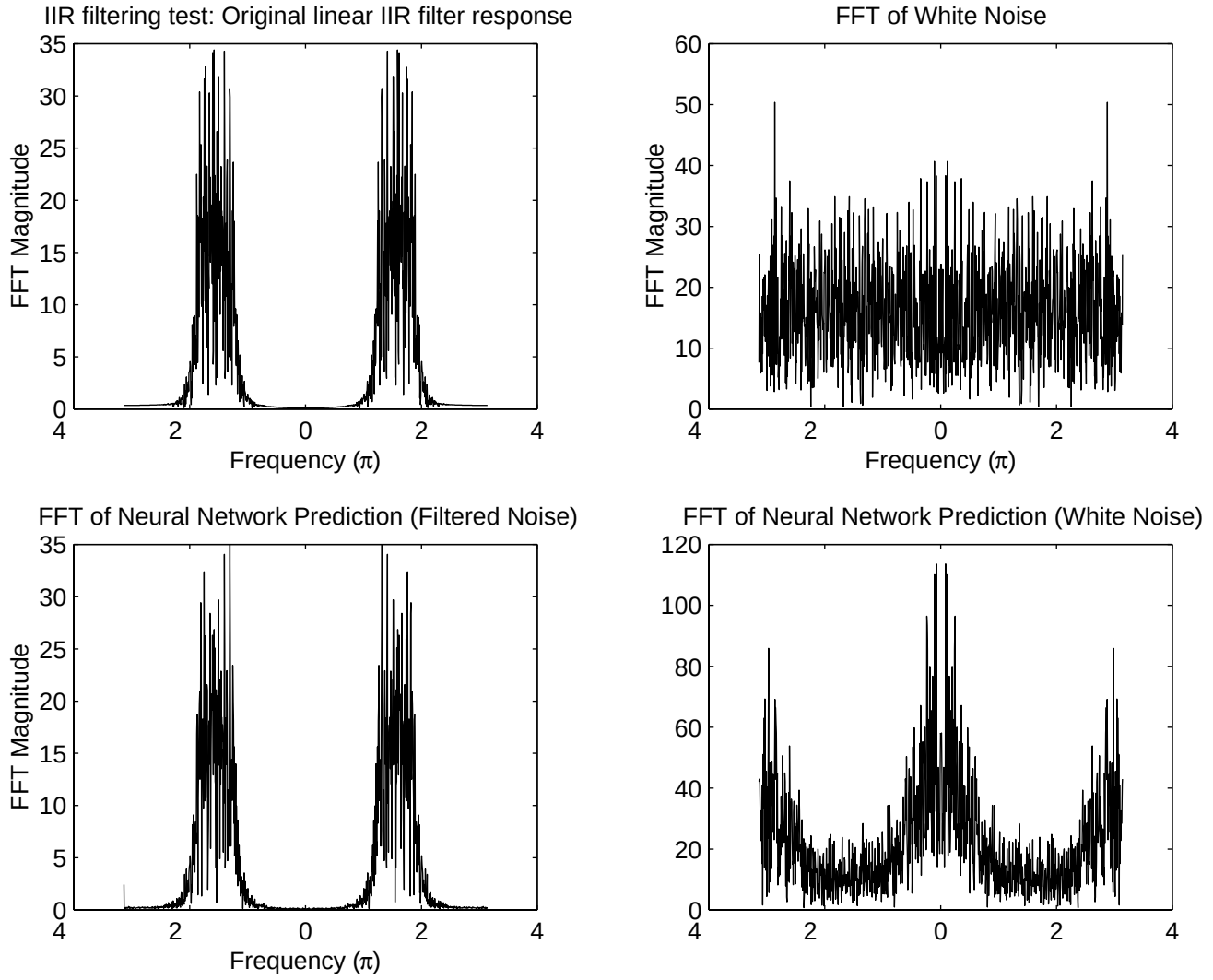

Figure 4: Comparison of various spectra. Note that the network does not perform correctly with white noise input.

have an IIR handy, we are faced with a difficulty. We cannot simply feed the network filtered noise values as we did in part 1 of the experiment; doing so re-interprets the network as an FIR filter, and we will get results entirely inconsistent with the task we wish the network to perform. The results of doing so are plotted in the rightmost quadrants of Fig. 4, in congruence with Fig. 2. Note that the frequency response of the network (as an FIR filter!) is not at all what we desire.Instead we must use the network as an IIR filter, that is, feed it its past predictions, as indicated in section 2.2.

In previous simulations, the network was presented its inputs in batch mode, which is the most efficient method in MATLAB for simulating such networks. However, in this case we do not know the inputs until we know the network's prediction for the previous sample. Writing the code to simulate the network in MATLAB proved too time-consuming; MATLAB is simply not constructed to perform iterative loops quickly.

However, there is an alternative method. MATLAB has a companion simulation engine, SIMULINK, which is designed to run iterative processes such as this quickly and accurately. The Neural Network Toolbox also has a generator `gensim` which will produce a SIMULINK block representing a network to be simulated.

The block diagram resulting is depicted in Fig. 5. Note that we can adjust the gain on the scale-factor G as a function of time, allowing for the creative use of the transformation represented by the network.

The scale-factor was set according to eq. (7), and the network simulated with a white-noise signal. The

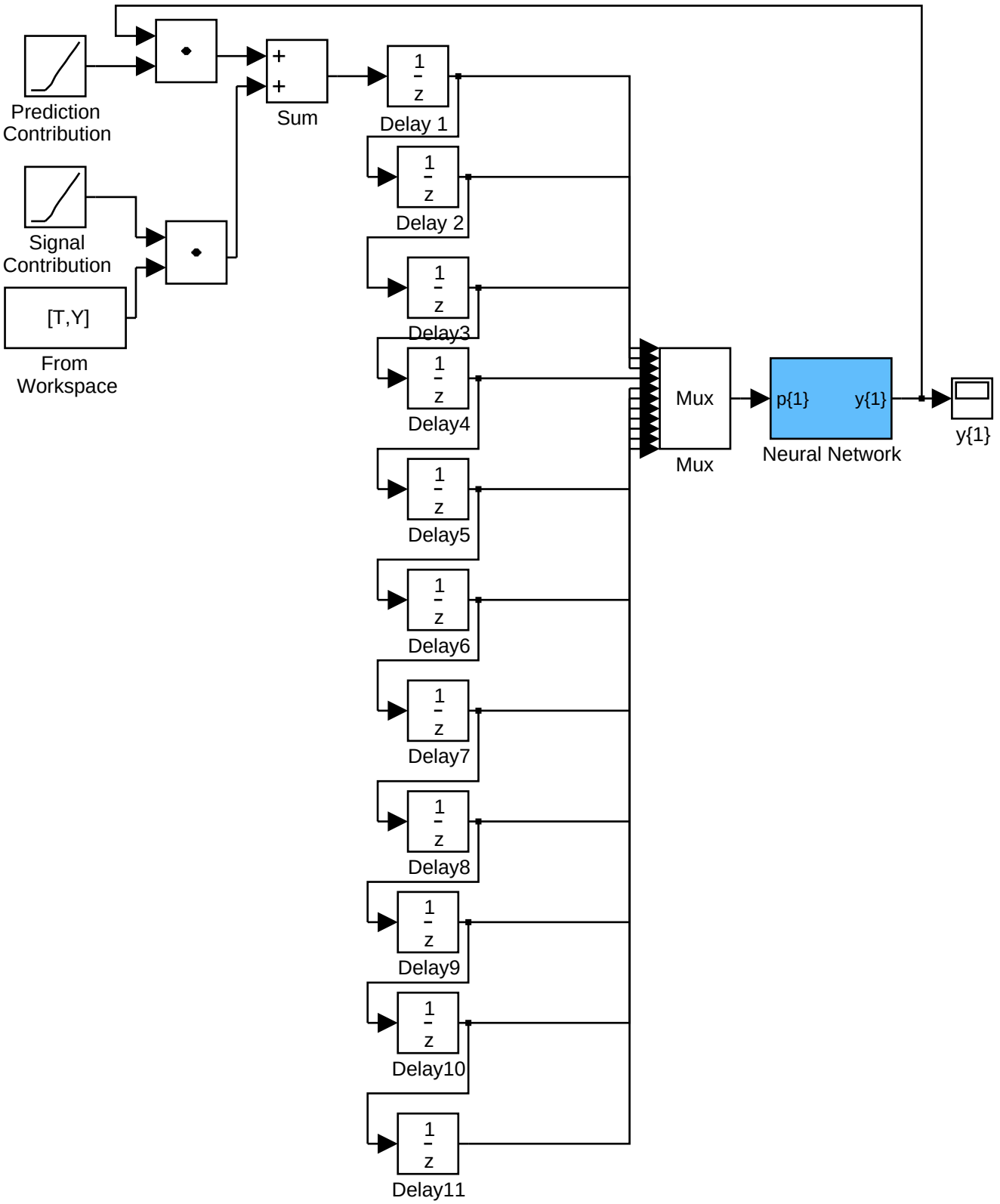

Figure 5: SIMULINK block diagram for the network trained for the IIR task

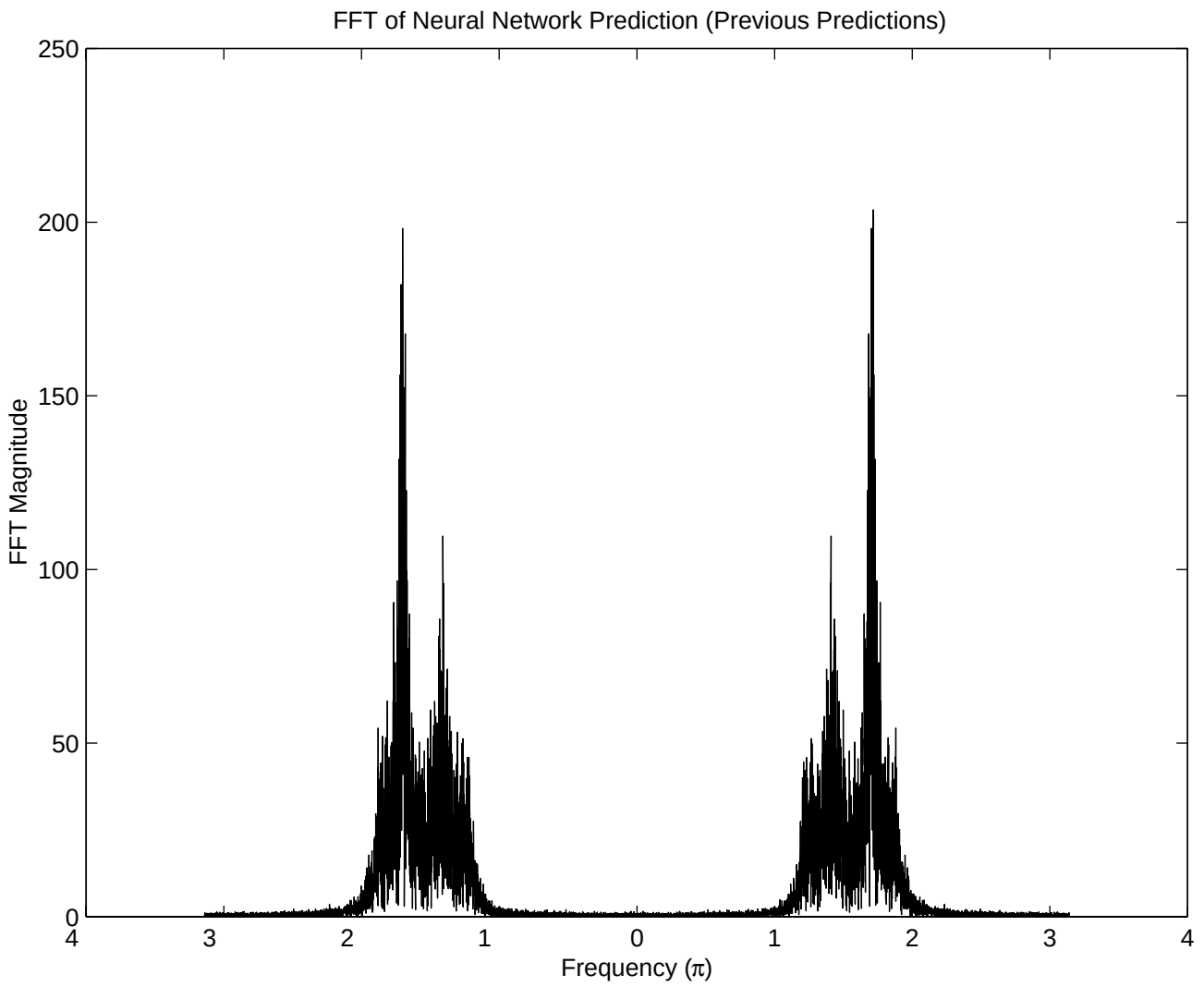

Figure 6: Spectrum of neural network using its own predictions as the generating data

results are plotted in Fig. 6. Note that the frequency response is now much closer to the original IIR filter response.

Note also the strong peaks in the middle of the bandpass region. This appears to be the result of the network overfitting the data (compare with its simulation on the original training set in Fig. 4). These strong peaks seem to be a feature of all networks studied; they tend not to generate flat responses.

4.3 LPC Learning Task

Finally, the neural network was trained with a "real" signal: a section of the chorus to Handel's *Messiah*. The song fragment, sampled at 8192 Hz, comes with the standard MATLAB distribution and can be loaded with the command `load handel`. Again the training set consisted of 1000 samples, here windowed to provide the 20 previous samples.

The error bound was somewhat arbitrarily chosen, as we do not have the luxury of an analytical model for the signal. The bound was a fraction of the total energy of the 1000-length sample, chosen to converge a reasonable number of epochs after the training algorithm's error minimization seemed to have reached its plateau. The training error is plotted in Fig. 7.

In this case it is difficult to ascertain simply from the error curve whether the network's learning was

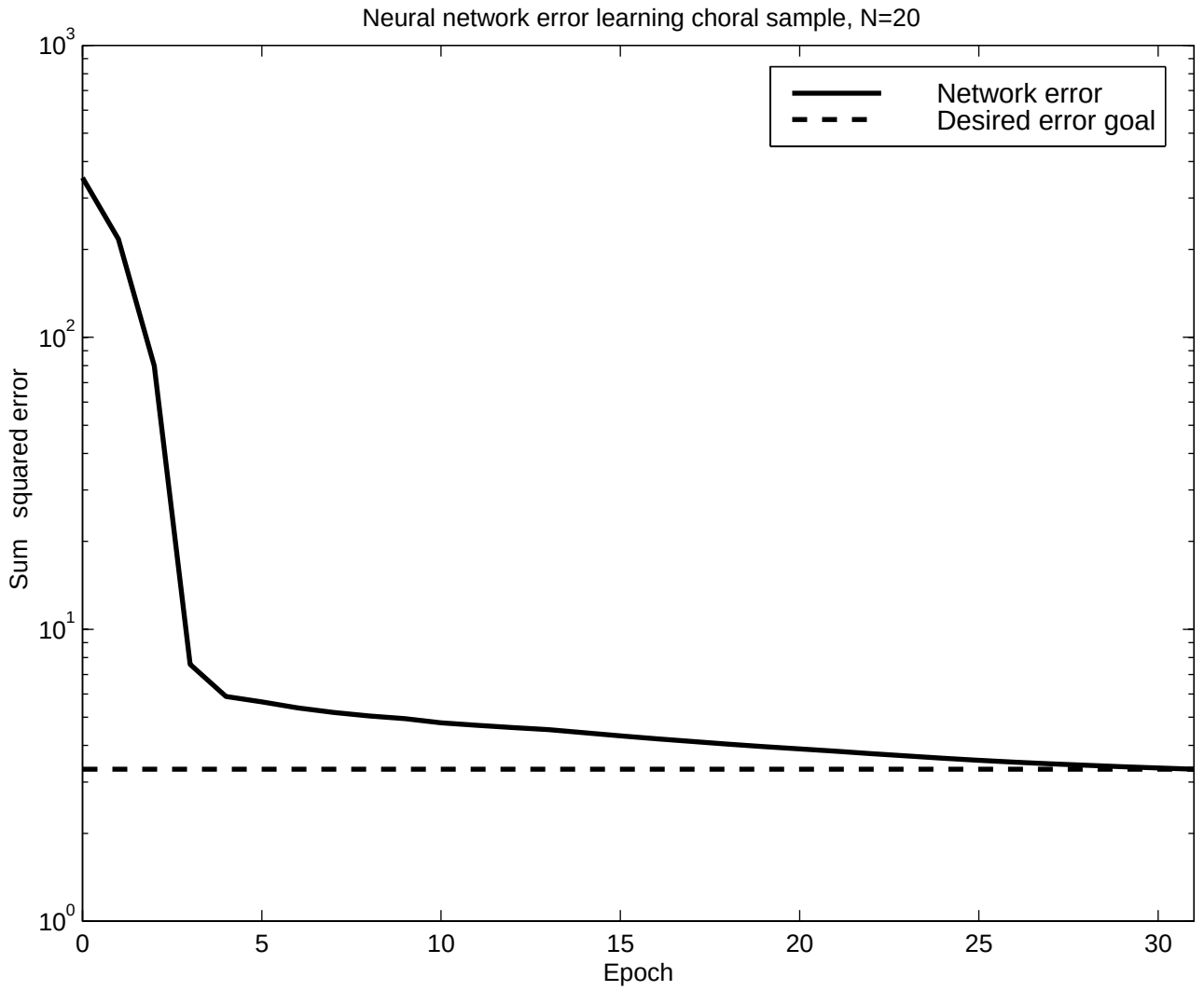

Figure 7: Error curve for vocal sample learning task

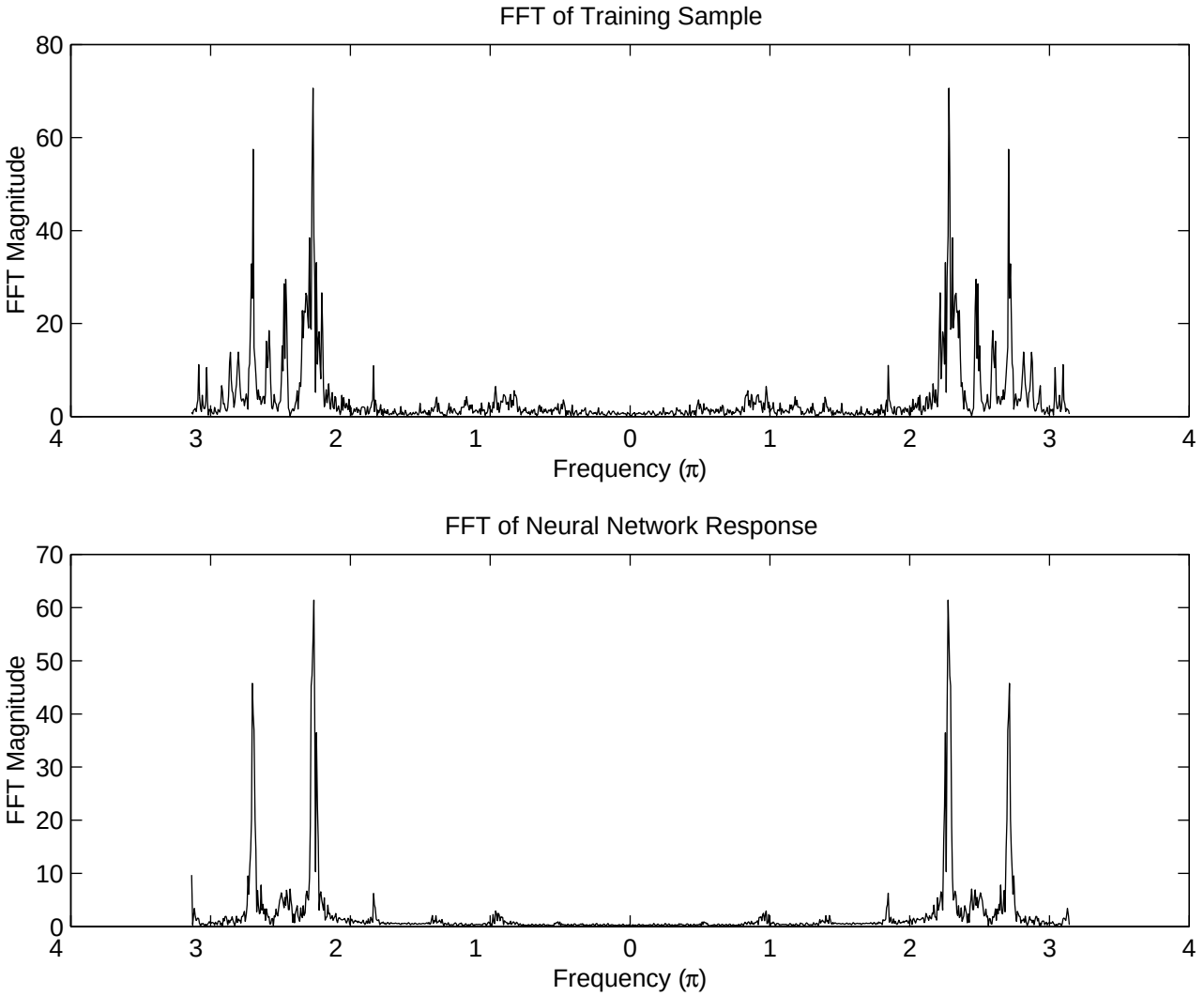

Figure 8: Comparison of spectrum for training data and neural network feedback response

"successful". The network was therefore plugged into a SIMULINK block diagram analogous to that shown in Fig. 5 (though in this case with more delay lines; N=20 to be precise). The scale factor G was again set according to eq. (7). The FFT of the original training sample is compared with that of the network prediction signal in Fig. 8.

Note that the frequency response of the network is a simplified version of that in the original signal. Note again the strong peaks, and the valleys where unfitted peaks were cut off. The trained network can be simulated for much longer than 1000 samples; the resulting sound does in fact have quite a strong vocal quality to it, and maintains the subjective pitch of the original.

5 Conclusions

A feedforward network was successfully trained to emulate three linear signal processing operators: a finite impulse response filter, an infinite impulse response filter, and finally a linear predictive coder. At each stage the network performed satisfactorily in relation to its linear predecessor.

The results of this simulation are promising. The trained network retains many of the favorable qualities of the equivalent LPC model. So far the network has not been tasked with anything not possible in the linear model;

it might be argued that nothing has been gained. However, the resulting representation offers opportunities for modification not possible in the linear domain. For example, the network parameters can be altered in realtime to change its response in a nonlinear fashion. The parameters could be swept between values appropriate for two separate training tasks, providing for a nonlinear interpolation between timbre. Other possibilities exist which will need to be left for future work.