Experiments Using Neural Networks to Invert the Public1 Model

Andrew Grauch

May 7, 2001

1 The Pulbic1 Problem

The publik1.f fortran program accepts as inputs the extinctions, scatterings, and asymmetry factors for 29 different cloud layers in the atmosphere as well as solar zenith and albedo measurements. From this data the program produces frequency readings. Our goal is to invert the public1.f model by training a neural network to map frequency data to extinctions and scatterings.

1.1 Why use a Neural Network

The public1.f model can be viewed as some function of the form I * T = O, where I is the set of inputs to public1.f, O is the set of outputs from public1.f , and T is a transformation that maps inputs to outputs. Finding the inverse of such a function involves finding a T' such that O * T' = I, where T' is a transformation that maps the outputs of public1.f to its inputs. We can easily compute T' using Matlab's psuedo-inverse function.

Figure 1.1 shows the results of the transformation, O * T' = I. Figure 1.1 shows the actual input to public1.f which is the desired output of the transformation. Clearly, the output of the transformation, O * T' = I, does not match the desired output. We can conclude that public1.f does not represent a simple linear transformation. It is still useful to think of public1.f as a function, though a non-linear one. Since neural networks can be used to approximate non-linear functions, a neural network seems justified as a means for inverting public1.f.

2 Experiment 1: Complete Data

2.1 Data Generation

We began our experiments by generating a set of random cloud data according to the following rules. Solar zenith is a random value between 0 and 89 inclusive. Originally the zenith was chosen randomly between 0 and 90, but zenith values of 90 produced a failure in the public1.f model due to division by the 0, cos(90). Albedo is chosen randomly between 0 and 1 inclusive. Each data set has a random number of 0 to 5 cloud

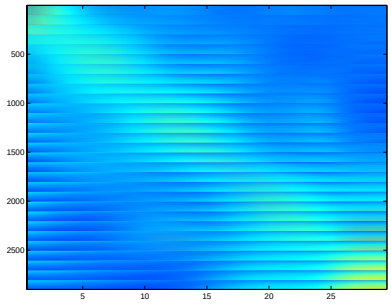

Figure 1: A linear transformation of the public1.f output computed using T' = inverse(input)*output

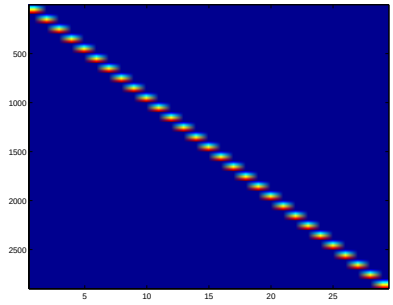

Figure 2: Actual inputs to public1.f, the expected outcome of the linear transformation

layers, which are distributed randomly between the 29 possible layers. Extinctions for each cloud layer were chosen randomly so that no extinction is greater than 10 and the sum of all the extinctions is not greater than 20. Scatterings for each layer are identical to the extinction for the layer. The asymmetry value is 0.85 if the layer has clouds, otherwise it is 0.

After each set of data was generated, public1.f was run and both its inputs and outputs were stored in a matrix for use by a neural network. The initial random random data contained 1000 different sets of cloud data.

2.2 Experimental Results

Dr. Charles Anderson has written some Matlab classes that implement a neural network. In a separate Matlab function we created a neural network from these classes and trained it using our random data. The data was broken into 3 groups. One group of 800 sets of cloud data was used for training and two groups of 100 sets of cloud data were use for validation and testing respectively. Initially the network was trained for 1000 epochs. For this test we varied the number of hidden units in the network between 20 and 40, the learning rates between 0.01 and 0.0001 for the hidden units and 0.1 and 0.001 for the output units, and the number of training epochs. We found that the network was unable to satisfactorily map public1.f outputs to inputs as indicated by high training, validation, and testing error rates.

3 Experiment 2: Constrained Data

3.1 Data Generation

With the desire of finding a more suitable mapping, we decided to constrain the problem. Rather than totally random data, we limited the output to a single layer of clouds in each data set. We generated the constrained data as follows. 100 sets of data were generated for each of the 29 different possible positions for a single cloud layer. Within each group of 100, there are 10 sets of data for each of the 10 possible extinctions. Each group of 10 contains every possible pairing of 5 different zeniths with 2 different albedos. The zeniths and albedos are fixed at even intervals in there respective domains.

As before as each set of data was generated, we ran public1.f and stored the inputs and outputs in a matrix for use by a neural network. The constrained data contained 2900 sets of cloud data.

3.2 Experimental Results

To train the neural network, we began by randomly permuting the order of the data. This was done to insure that the network could be validated and tested with data from random points in the domain rather than from a fixed point in the domain. Our data was ordered by cloud layer and we felt it inappropriate to to always validate and test using data from the same cloud layers. 2320 sets of data were used for training and 290 sets were used for both training and validation.

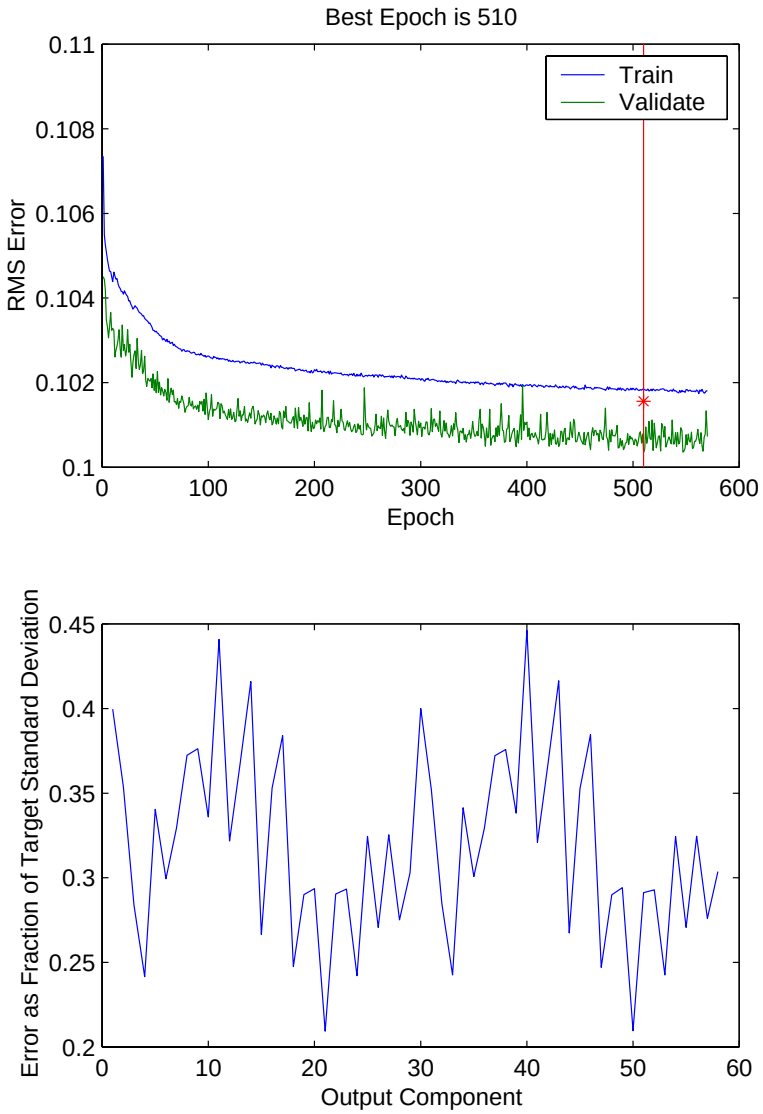

Figure 3: Best network trained on constrained cloud data

Using the constrained data we trained a neural network with 20 units in the hidden layer for 2000 epochs. We found that both the network's error rate and validation rate approached 0.1 as training progressed. However, the difference between the error and validation rates remained high. We tried again using 40 hidden units with similar results. We also varied the learning rates of the network, again with similar results.

Figure 3.2 shows the training and validation error curves for the best performing neural network in this set of tests and the error in the output component. The top graph also shows the epoch for which the test error was the lowest. This network contained 40 hidden units and was trained with learning rates of 0.00001 and 0.001 for the hidden units and the output units. We note that the greatest amount of learning occurred during the first few epochs.

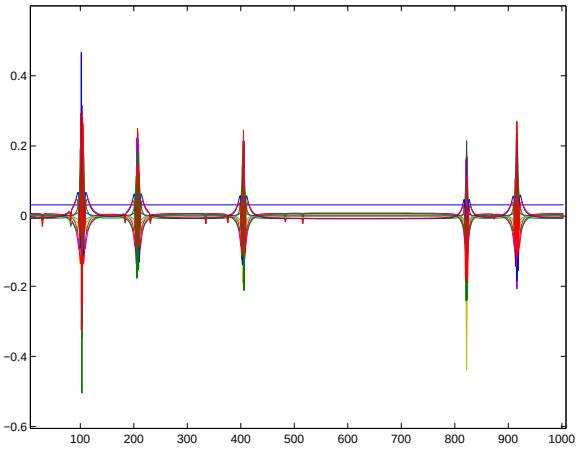

Figure 4: The eigenvectors associated with the 10 largest eigenvalues

4 Experiment 3: Reduced Data

4.1 Data Generation

Still desiring to find lower error rates, we again chose to restrict the scope of the training problem. In the previous two tests we were attempting to map 1003 inputs onto 58 outputs. We argue that by reducing the number of inputs we may find a better mapping. We reduced the 2900 sets of data that we had previously generated. We viewed the inputs to the network as a 2900 X 1003 matrix. To reduce the matrix of inputs we began by finding its convariance, the eigenvalues, and the eigenvectors of the resulting matrix. Matlab has built in functions to accomplish these tasks. The eigenvalues and eigenvectors were stored as separate matrices. We then picked the 10 most significant eigenvectors. We argue that these vectors contribute the most to the function that describes the matrix of inputs. In this case the chosen eigenvectors were all associated with the largest non zero eigenvalues of the covariant matrix. They were chosen by inspection of the diagonal of eigenvalue matrix, but in the future we could find the non zero values programmatically. Figure 4.1 shows a sampling of the outputs of the Public1.f model which are then input into the neural network. Figure 4.1 shows the eigenvectors we picked. The eigenvectors contain information similar to the frequency spikes seen in the Public1.f output. If our hypothesis about the eigenvectors is correct, then by transforming the Public1.f outputs we can compress the inputs to our neural network without a serious loss of information. To transform the inputs, the 10 vectors were made into a 2900 X 10 matrix. We projected our inputs onto the space defined by this matrix. The results of the projection and the desired outputs were then combined into a separate matrix for use by a neural network.

4.2 Experimental Results

As in the previous test, we randomly permuted the order of the data. 2320 sets of constrained and reduced cloud data were used for training and 290 sets were used for validation and testing respectively. We trained a neural network with 20 hidden units for 1000 epochs. For the first trials using this data, we used high learning rates in the network. We had hypothesized that with fewer inputs the network could quickly learn a mapping. We were wrong. The network was not able to find a mapping using high learning rates. Figure 4.2 shows typical output from networks trained with high learning rates. The network used to generate this particular output used 20 hidden units and was trained with a hidden unit learning rate of 0.01 and an output unit learning rate of 0.1. The continual fluctuation in the training errors indicates that little or no learning occurred.

We then tried lowering the learning rates by a factor of 100. With these rates the network was able to learn a mapping similar to our prior tests. Both validation and training error rates approached 0.1 as training progressed. However, the difference between the training and validations errors was reduced to

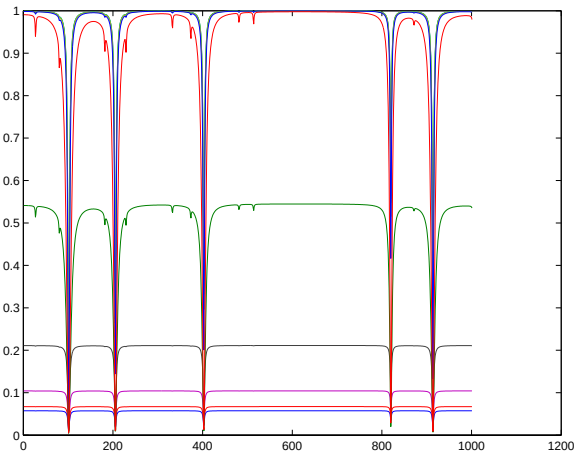

Figure 5: Output of Public1.f, Input to the neural network

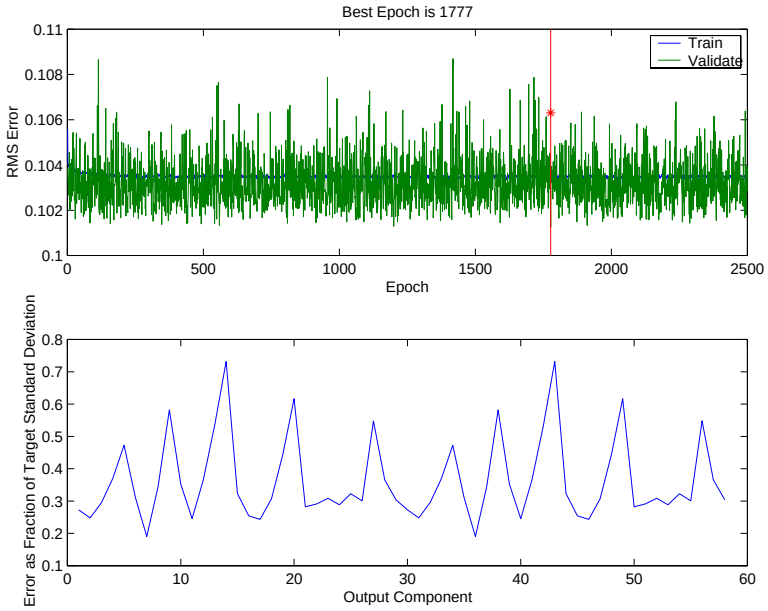

Figure 6: Typical output from a network trained to the reduced data using a high learning rate

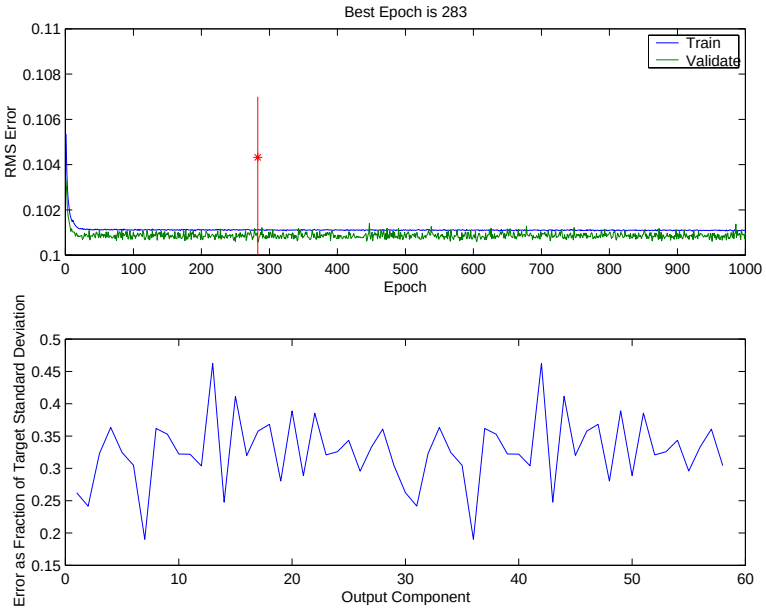

Figure 7: Best network trained on reduced constrained cloud data

the point of being almost indistinguishable. Figure 4.2 shows the best results of network trained using low learning rates. The network contained 40 hidden units and was trained using a hidden unit learning rate of 0.00001 and an output unit learning rate of 0.001.

5 Experiment 4: A Different Feedback Function

5.1 Generating Data

All previous tests were performed on neural networks whose nodes used a tangent threshold function. Anderson's neural network code supports the use of a sigmoid threshold function. This test examined the effect of using of the sigmoid function. Data for this test was generated in the same manner as the previous test. The constrained cloud data was reduce in the manner described above and randomly permuted for testing.

5.2 Experimental Results

A single test was performed with good results. The sigmoid function caused the network to produce training and validation error rates that were approximately half the magnitude of error rates from the tangent feedback function. The error rates approach 0.056 as training progressed. The error as a fraction of standard deviation rates were also lower than those from networks using a tangent function. Figure 5.2 shows the result of the network trained with a sigmoid feedback function. The network contained 40 hidden units and was trained using a hidden unit learning rate of 0.00001 and an output unit learning rate of 0.001. Like the previous tests, the greatest amount of learning occurred in the first few training epochs.

The results from this test are very encouraging. This is the first test where we have seen error rates below 0.1.

6 Reproducibility of Matlab Results

After obtaining some good results using a neural network with Matlab, we asked if we could reproduce the results using different neural network implementations. One disadvantage of Matlab is the time required to train a network. Ideally, would like to find a implementation that runs quickly and produces high quality

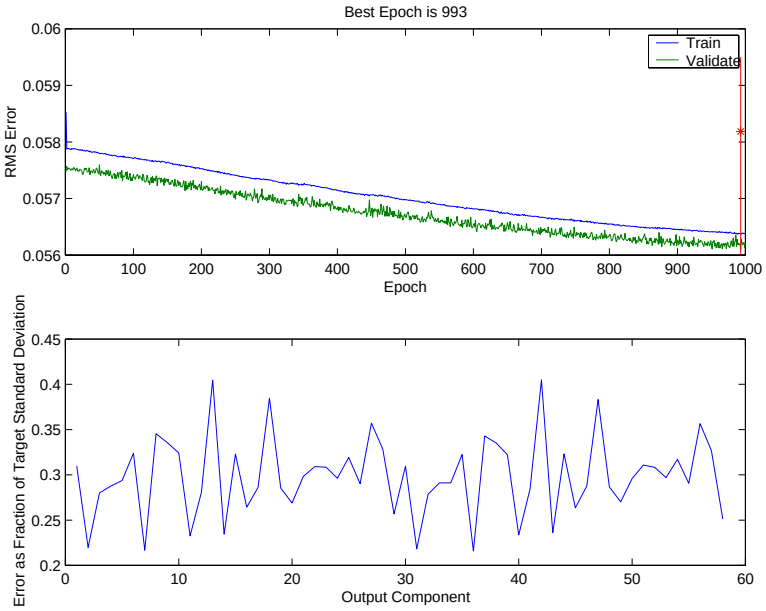

Figure 8: Output from a network trained using a sigmoid feedback function

results. Dr. Anderson obtained some excellent results with a network written in C. We also obtained similar results with a network written in Java. The remainder of this paper shall focus on the Java neural network.

Before training the Java neural network, the data we generated in Matlab had to be written to ASCII files; and, the Java network needed to be modified to read training, validation, and testing data from the ASCII files.

The data set used to train the Java network was significantly different from that used to train the Matlab network. We standardized the distribution of data. Matlab networks were trained using randomly ordered data, theoretically not all cloud layers were represented in the training set. By insuring that the data is well distributed, we guaranteed that all cloud layers were equally represented in the training, validation, and test sets. We reduced the data by projecting onto the fifty best eigenvectors rather the best ten.

After training we compared the results with the Matlab implementation. There were considerable differences between the two networks, undoubtedly due to the differences in the data sets. The main difference between the two networks is the perceptron transfer function. The best results in Matlab were found using a sigmoid transfer, but in Java the best results were found with a hyperbolic tangent transfer. The learning rates and basic architecture also differed.

The best Java network results were found using a network with 80 hidden units, a hidden learning rate of 0.01, and an output learning rate of 0.001 that was trained for 10000 epochs.

The average error in each network unit, the root mean error, was 0.0673, which is very comparable to the error in the best Matlab network.

7 Experimental Conclusions

Using the best Java network, we were able to test the hypothesis: a neural network can successfully invert the public1.f solar flux model, where success is measured by the number of cloud layers correctly identified by the network.

When inverting the public1.f model we want the network to identify cloud layers given a set of frequency readings. Figure 7 shows an example of desired network output, the cloud layer occurs at the spike in the graph. The network correctly identifies a cloud layer when its maximum output occurs at the same place as the spike.

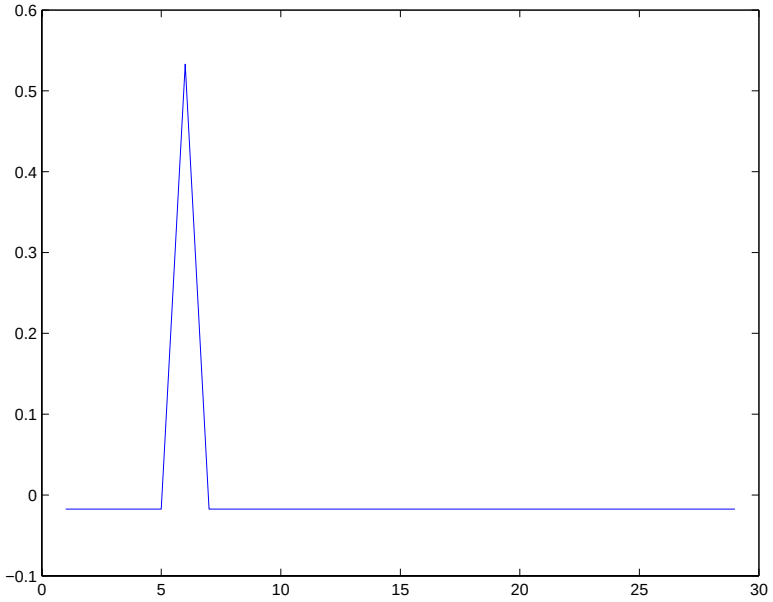

Figure 9: Desired network output: A single cloud observation

After training was completed, the network was presented with 290 unseen testing cases. The network correctly identified 167 of these 290 cases. The following figures show some correctly identified cloud layers.

Figure 7 shows a misclassified test case. Many misclassifications occurred when the magnitude of the cloud layer was small enough to get lost in the general noise of the network output.

The test set had 10 cases from each of the 29 different cloud layers. Figure 7 shows the number of cases correctly classified by cloud layer. The network was able to better classify cases from the lower layers than those from the upper layers. The network correctly classified 83 of the 100 cases in the first 10 layers, but only 84 of the remaining 190 cases.

There are 10 distinct values, ranging from 0.07 to 0.9, for the magnitude of the spike indicating the cloud layer. Figure 7 shows the number of cases correctly classified by value of the cloud layer. The network had difficulty classifying layers identified by small values, due to noise in the output.

When the network correctly identified the cloud layer, there was some error in the predicted and expected values of cloud magnitude. The error varied from 0.0002 to 0.7. Figure 7 shows the number of correct classifications by the error in the maximum values.

From these results, we can conclude that our hypothesis is at least partially correct. Neural networks can invert the public1.f model on a very limited range of data. The limitations we have identified are: cloud predictions are only accurate for a single cloud layer; low cloud layers are more easily identified; more intense cloud layers are more easily identified; cloud predictions are only valid for a small set of zenith and albedo readings. These limitations suggest that further research is needed.

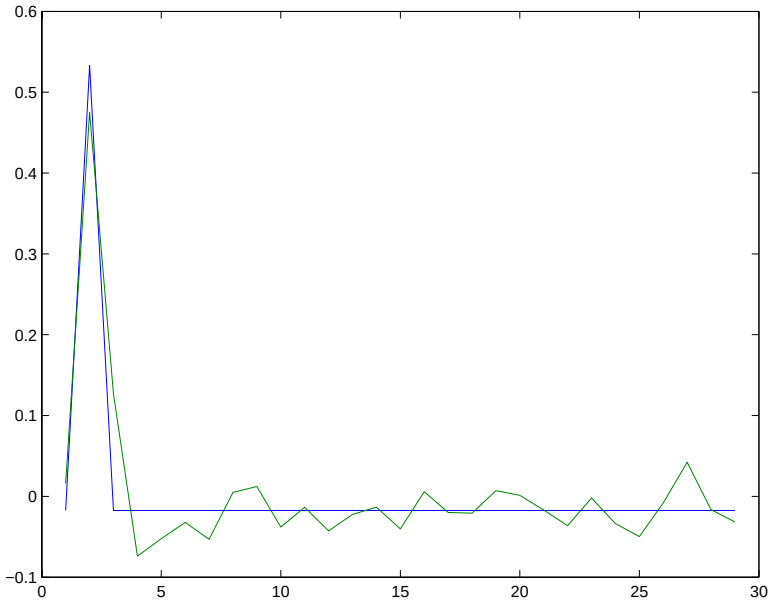

Figure 10: Correctly classified case 1

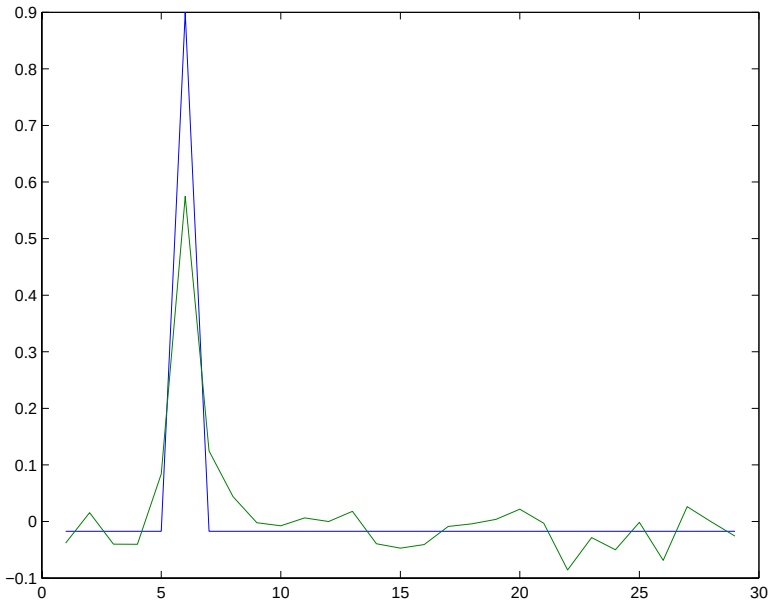

Figure 11: Correctly classified case 2

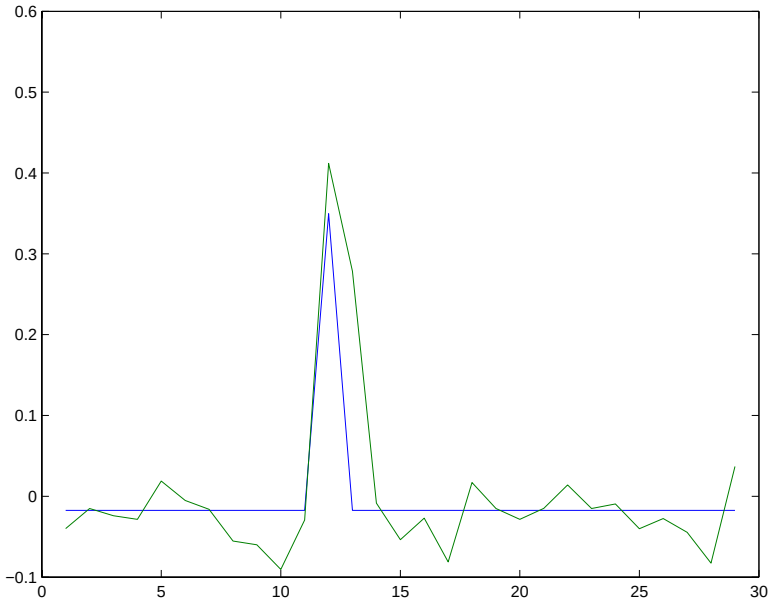

Figure 12: Correctly classified case 3

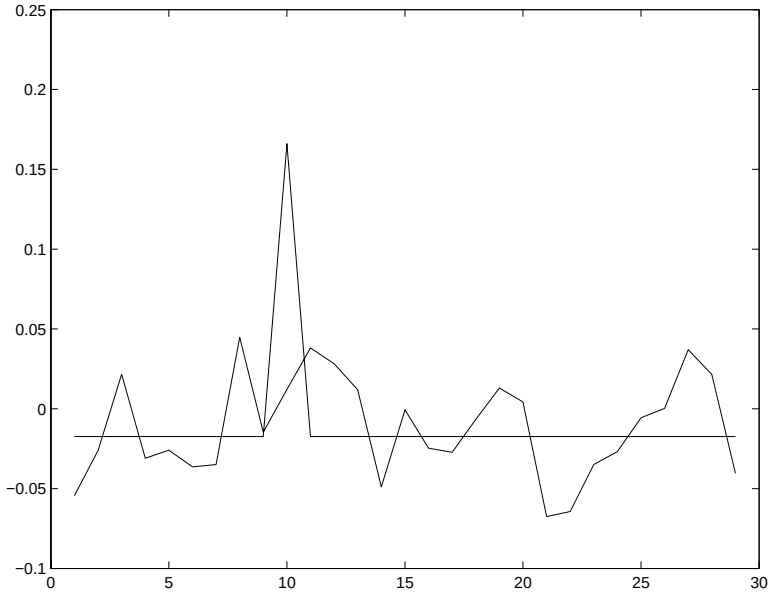

Figure 13: A missclassified test case

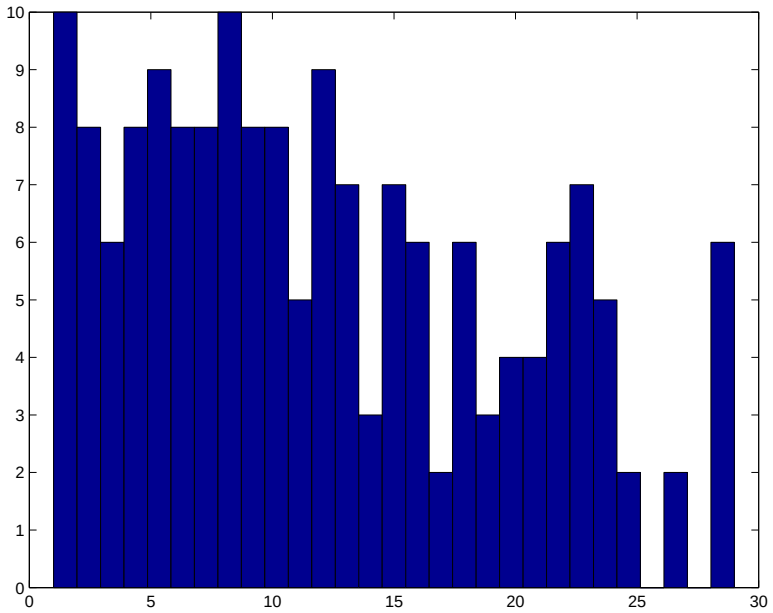

Figure 14: Number of correctly identified cases by cloud layer

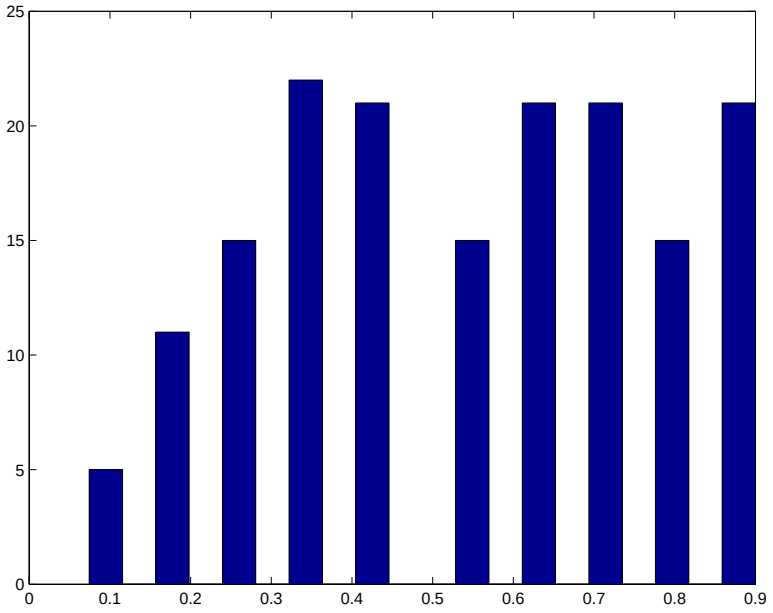

Figure 15: Number of correctly identified cases by cloud magnitude

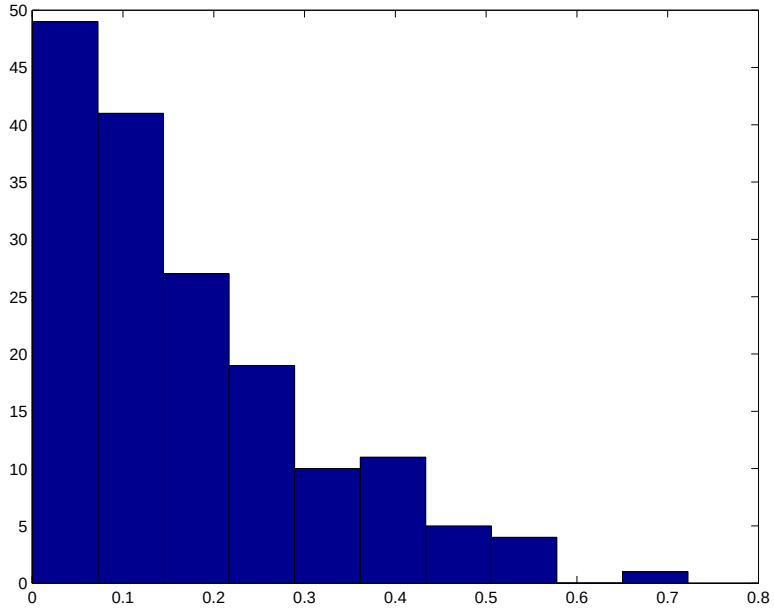

Figure 16: Number of correctly identified cases by error in maximum values