Using a Two-Layer Neural Network and Physicochemical Properties to Classify Glass for Forensic Analysis

Michael Arango
mikearango@gwu.edu

Mark Barna
mark.barna@gmail.com

Paul Brewster
pfbrewster@gmail.com

June 30, 2017

Contents

1 Project Proposal

In this study, we will use the physicochemical properties of glass to determine whether or not a given glass sample was taken from a window. This is a fundamental problem in forensic analysis as it is highly unlikely that glass fragments will be found on people unless they have been present at the time glass breaks. Glass analysis is of vital importance in forensic science as it allows us to test if the glass fragment found on a person is the same as the glass at a crime scene. Since glass is made up of several raw materials and certain elements impart specific properties, we can find out a lot about the glass if we analyze the chemical composition.

The dataset we chose for analysis was made available for download from the UCI Machine Learning Repository and was created by the USA Forensic Science Service. There are 214 observations of 9 different features along with 214 targets that specify whether the glass sample came from a window or not. While we would like more data to train a neural network, we believe the dataset is large enough for our purposes. It is difficult to know before we train a neural network if we have enough data, but the amount of data required is directly related to the complexity of the underlying decision boundary we are trying to implement. We won't know how complex the decision boundary we are trying to approximate is until we train the network, but we feel confident using the dataset as many others have used the dataset and found robust results. Several other papers in the literature use much more complex methods than we will employ and have not found the size of the data to be an issue.

We have chosen a two-layer perceptron network with tangent-sigmoid transfer functions in the hidden layer and *softmax* transfer functions in the output layer. This is a fairly standard network for pattern recognition. Moreover, we will use the *Scaled Conjugate Gradient (SGD)* algorithm to train the network as it is good for pattern recognition problems in which the output layer uses a non-linear transfer function. Since we do not expect the training error to converge to zero, we implement early stopping criteria to prevent overfitting. Lastly, we use *cross-entropy* as our performance index since our targets take on discrete values and it is the optimal performance index for pattern recognition networks that use the *softmax* transfer function in the output layer.

Two different frameworks will be used to implement the neural network. First, we will use the Neural Network Toolbox, specifically the Neural Network Pattern Recognition Tool (`nprtool`) train, validate, and test our network. We use this framework to start with a simple graphical user interface to quickly ensure our specified network architecture is appropriate and to get baseline performance statistics. Then, we will replicate the analysis in Python to gain practical experience building network architectures in a scripting language. Note that since the goal is practical experience, we will not be leveraging the power of the *scikit-learn* package (`sklearn`) for this exercise.

Several reference materials will be consulted to obtain sufficient background knowledge of the subject at hand. First, we plan on doing a thorough review of the forensic chemistry and geology literature to understand the reasons for using physicochemical properties to classify glass. Then, papers on glass analysis will be examined to supplement background knowledge with experiential knowledge.

Considering our problem is one of pattern recognition, a confusion matrix will be used to assess the accuracy of our model and the *false postive* (Type I error) and *false negative* (Type II error) rates. Further, the *Receiver Operating Characteristic (ROC) curve* will be used to compare the true positive rate to the false positive rate. This will help us gain additional knowledge of the predictive power of our network.

We plan to finish our research and submit it by Wednesday, June 28, 2017.

2 Literature Review

The dataset we will use for this analysis was initially employed in Evett and Spiehler's paper *Rule Induction in Forensic Science* (1987) [ES87]. They recognized the usefulness to a forensic crime lab of classifying glass fragments based on refractive index and chemical composition. This would, for example, allow the lab to ascertain whether samples gathered on a suspect?s clothing came from a window, potentially indicating they had broken it, or from another source, like a broken bottle.

In their experiment, Evett and Spiehler wished to see if the Bionic Evolutionary Algorithm Generating Logical Expressions (BEAGLE) machine learning algorithm could correctly classify the glass—first as either window or non-window, and then into a second level of sub-categories. Our project focuses on the former and leaves the latter as a future exercise. The BEAGLE algorithm Evett and Spiehler used to train the network uses a series of logical "and" statements to chain rules together based on the inputs. They offer up the following example of a rule:

$$\{(\text{Fe} \leq \text{Na}) \text{ and } [\text{K} > (\text{Fe} \cdot 650)]\}, \tag{1}$$

where Fe, Na, and K are the percent composition of iron, sodium, and potassium, respectively, of each glass sample. Evett and Spiehler found that the BEAGLE algorithm outperformed the k-Nearest Neighbors algorithm (with $k = 3$) and Linear Discriminant Analysis (LDA)—the models they used for baseline performance measures.

The Department of Justice regularly issues research grants for the elemental analysis of glass. In 2012, they issued one such grant to researchers at Florida International University (FIU) to work with the Miami-Dade County Police Department [AT12]. The researchers note that analysis of small quantities of materials has become an important yet underutilized type of evidence at many crime scenes including hit-and-run accidents

and other violent crimes. The ability to classify different types of glass could be of vital importance in the case of a hit-and-run. Further, the group of researchers attempted to compare the discrimination power between the methods used in most forensic laboratories for glass analysis. Their aim was to create a more "standard" method that can be used by the operational forensic laboratory and a "match criteria" for use in routine casework situations.

Maureen Bottrell, a geologist and forensic scientist at the FBI Laboratory released a report in 2009 documenting the background information that ought to be used when comparing glass samples with data [Bot09]. She notes that the vast majority of raw materials used to make glass are derived geologically and that North American glass makers use more than 20 million tons of raw materials annually. All of these materials contain several impurities that result in perceived differences in glass products.

Bottrell writes that physical properties such as color, curvature, fluorescence, thickness, and surface features should first be used to determine if the material fragments are glass. Once we know a sample is glass, Bottrell recommends using optical properties, particle immersion, density, and elemental analysis to differentiate between types of glass. In this study, we focus on optical properties, specifically the refractive index, and elemental analysis to classify whether glass samples came from a window or not.[1]

Since glass is made up of several raw materials and certain elements impart specific properties, we can find out a lot about the glass if we analyze the chemical composition. Glass made on the same manufacturing line over a period of time can often have highly variable properties as mixtures of raw materials can have drastically different chemical compositions.

3 Description of the Dataset

The dataset was made available for download from the UCI Machine Learning Repository and was created by the USA Forensic Science Service [MA94]. The purpose of this dataset is to use physicochemical properties to classify whether a certain glass fragment comes from a window or not.

3.1 Inputs

The matrix of inputs contains 214 observations of 9 variables and there is no missing data. Of these variables, eight of the nine measure the percent weight that a given elemental oxide makes up of the total glass sample weight. All the eight elements except silicon are classified as metals on the periodic table of elements. Sodium and potassium are alkali metals whereas magnesium, calcium, and barium alkaline earth metals. Aluminum and

[1] See section 3.1 for more on the refractive index.

iron are classified as poor metals and transition metals, respectively. The last variable in the input matrix represents the refractive index which measures the speed of light in a transparent medium and is known as Snell's law. It can be represented formulaically as the ratio of the velocity of light in a vacuum to the velocity of light in the glass itself: $n = \frac{c}{v}$. A more thorough description of target variables is as follows:

Refractive Index: measures the ratio of the velocity of light in a vacuum to the velocity of light in the glass itself

Sodium: represents the percent weight in sodium oxide (Na_2O)

Magnesium: represents the percent weight in magnesium oxide (MgO)

Aluminum: represents the percent weight in aluminum oxide (Na_2O)

Silicon: represents the percent weight in silicon oxide (Al_2O_3)

Potassium: represents the percent weight in potassium oxide (K_2O)

Calcium: represents the percent weight in calcium oxide (Ca_2O)

Barium: represents the percent weight in barium oxide (Ba_2O)

Iron: represents the percent weight in iron oxide (Fe_2O_3)

3.2 Targets

The matrix of targets has 214 observations, one for each observation in the training set, where a given target is denoted by $\begin{bmatrix}1\\0\end{bmatrix}$ if the glass sample comes from a window and $\begin{bmatrix}0\\1\end{bmatrix}$ otherwise. Note that the targets are two-dimensional instead of the more common one-dimensional binary encoding. This two-dimensional encoding allows us to have only one neuron firing at a time and tends to result in marginally better performance.

4 Description of the Network Architecture and Training Algorithm

Once the data is preprocessed, the next step is to decide on or create a network architecture. The basic network architecture is determined by the problem we wish to solve. Once the basic network architecture is determined, we decide how many layers, how many neurons in each layer, how many outputs the network should have, and what kind of performance index function we should use for training [Dem+14].

4.1 Network Architecture

The standard neural network architecture for pattern recognition problems is the multi-layer perceptron with tangent-sigmoid transfer functions in the hidden layers and *softmax* transfer functions in the output layer. For most problems, including a fairly simple one like ours, one hidden layer usually suffices. Thus, we will implement a two-layer perceptron. If the results of the network are unsatisfactory after training and testing with one hidden layer, we will retrain with an additional hidden layer, but we do not anticipate having to do this. The *tansig* transfer function is usually preferred to the *tansig* transfer function in the hidden layers since it produces outputs (which are inputs to the next layer) that are centered near zero, whereas the *tansig* transfer function always produces positive outputs.

We also need to select the number of neurons in each layer. The number of neurons we use in the output layer should be the same as the size of the target vector. In our case, this means we should use two neurons in the output layer. On the other hand, the number of neurons we use in the hidden layer is directly proportional to the the complexity of the decision boundary being implemented. Since we do not know the complexity of the decision boundary needed to classify these glass samples before training, we begin with ten neurons, which may be more than we need, and leverage early stopping techniques to prevent overfitting [Dem+14].

Now that we have chosen a network architecture, we can calculate the network output. The output from the hidden layer (the input to the output layer) can be calculated as

$$\mathbf{a^1} = \mathbf{tansig}(\mathbf{W^1 p} + \mathbf{b^1}), \tag{2}$$

while the output from the output layer is

$$\mathbf{a^2} = \mathbf{softmax}(\mathbf{W^2 a^1} + \mathbf{b^2}). \tag{3}$$

The network architecture can be seen in **Figure 1** below.

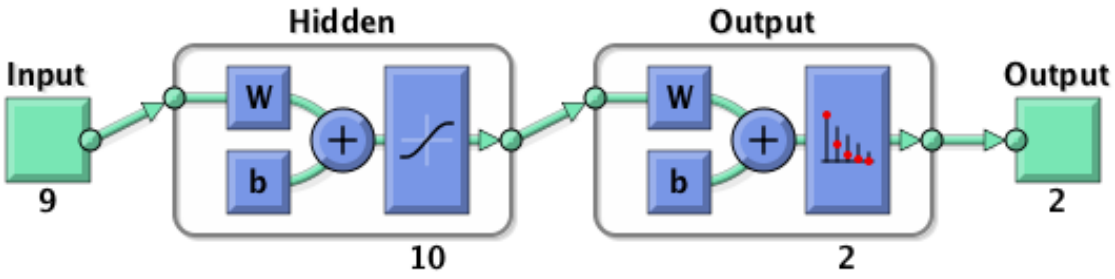

Figure 1: Two-Layer Perceptron Used to Classify Glass Samples

4.1.1 History of the Perceptron

Before discussing the training algorithm, we offer up a brief history of the perceptron. Much of the modern interpretation of neural networks is credited in large part to Warren McCulloch and Walter Pitts who showed that networks of artificial neurons could calculate any function. We still use the fundamental feature of their model in which a weighted sum of inputs is compared to a threshold to determine the output of a neuron [Dem+14].

One of the first applications of neural networks came in the 1950's when Frank Rosenblatt developed the perceptron network and the corresponding learning rule to solve pattern recognition problems. While his developments were monumental at the time, several researchers showed that a single-layer perceptron and the learning rule could not solve certain problems. Specifically, the error will never converge to zero when using the perceptron learning rule if the input vectors are *linearly inseparable*. It wasn't until the 1980's that multi-layer perceptron networks and more complex learning rules were proposed that could solve these problems.

Widrow and Hoff's Least Mean Square (LMS) learning rule suffered from the same disadvantage as Rosenblatt's, but it has since been generalized. The generalization of the LMS learning rule is referred to as *backpropagation* and we commonly use it to train multi-layer perceptron networks [Dem+14].

4.2 Training Algorithm

We chose to use the Scaled Conjugate Gradient (SCG) algorithm in MATLAB to train our network as it is very efficient for pattern recognition problems. For multi-layer networks, the Levenberg-Marquardt algorithm is often used, but it does not work well for pattern recognition as the transfer function in the output layer is operating outside the linear region. The scaled conjugate gradient algorithm is a special type of backpropagation. For the replication of the network in Python, we use standard backpropagation with steepest descent.

The implementation of backpropagation can be broken down into three steps:

1. Propagate the input forward through the network
2. Propagate the sensitivities backward through the network
3. Update the weights and biases using the approximate steepest descent rule

4.2.1 Forward Propagation

For a multilayer network, the output from one layer is the input to the next layer. That is, the hidden layer's output is the input to the output layer of the network. The neurons

in the first layer receive external inputs (the observed data):

$$\mathbf{a}^0 = \mathbf{p}, \tag{4}$$

which serves as the starting point for the network. After the input is received, it is propagated forward with the following equation

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}), \text{ for } m = 0, 1, \ldots, M-1 \tag{5}$$

where we use M to denote the number of layers in the network. Then, the output from the the neurons in the last layer are

$$\mathbf{a} = \mathbf{a}^M. \tag{6}$$

4.2.2 Performance Index

Before moving on to the step of backpropagating the sensitivities, we discuss the performance index as it is a critical part of the process. The performance index is a measure of the error of the network outputs in relation to the targets. The general implementation of the backpropagation algorithm uses the *mean square error* as a performance index. The mean square error is approximated by taking the expectation of the sum of the squared errors (residuals). However, we choose a different performance index to gauge our model's predictions.

Mean square error works very well for functions with continuous target values—the case where we are approximating a function. However, in pattern recognition we are given discrete target values, so other performance indices that take this into account make more sense. We choose to use *cross-entropy* as our performance index. This is a commonly used performance index when the *softmax* transfer function is used in the output layer. Given a set of input-target pairs

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\} \tag{7}$$

where $\mathbf{p}_q$ is an input and $\mathbf{t}_q$ is the corresponding target output, we denote the cross-entropy loss as follows:

$$F(\mathbf{x}) = -\sum_{q=1}^{Q}\sum_{i=1}^{S^M} t_{i,q} \ln \frac{a_{i,q}}{t_{i,q}} \tag{8}$$

where Q is the number of samples in the dataset and S^M is the number of neurons in the output layer. We can simplify this by vectorizing the operation over all input-target

pairs, $\{\mathbf{p}_q, \mathbf{t}_q\}$, and eliminating the first summation:

$$-\sum_{i=1}^{S^M} \mathbf{t}_i \ln \frac{\mathbf{a}_i}{\mathbf{t}_i}. \tag{9}$$

Recall that in our pattern recognition problem we have two classes where the targets are $\begin{bmatrix}1\\0\end{bmatrix}$ and $\begin{bmatrix}0\\1\end{bmatrix}$. Thus, each neuron can only take on values of zero or one. This allows us to further simplify the equation to

$$-\mathbf{t} \ln \mathbf{a} \tag{10}$$

since $\mathbf{t}_i = 0$ implies the cross-entropy loss for the i-th neuron is zero and the case where $\mathbf{t}_i = 1$ implies the cross-entropy loss is simply $-\mathbf{t}_i \ln \mathbf{a}_i$.

It is important to note the backpropagation algorithm works with any differentiable performance index we specify. We just need to change the initialization of the sensitivities in the output layer accordingly. This brings us back to the next step: propagating the sensitivities backward through the network [Dem+14].

4.2.3 Backpropagation

Once we have propagated the input forward, we tcompare the network output to the targets so we can use the error to adjust the weights and biases accordingly. But, in the case of a multi-layer network, the errors and the performance index we use to evaluate those errors are no longer just a function of the weights. Rather, they are indirectly a function of the weights in the hidden layer in addition to the weights in the output layer. Thus, we call the errors we propagate backward through the network 'sensitivities' because they represent the sensitivity of the performance index, $F(\mathbf{x})$, to changes in the i-th element of the net input at layer m. We can calculate these sensitivities analytically as follows:

$$s_i^m \equiv \frac{\partial F}{\partial n_i^m}. \tag{11}$$

Now, we compute the sensitivities $\mathbf{s}^m$. Note that the reason we call it backpropagation is because the sensitivities are propagated backward through the network via a recurrence relation where the sensitivity at layer m is calculated from the sensitivity at layer $m+1$. This is easier said than done, however, since we need to find a Jacobian matrix of sensitivities to derive the recurrence relation:

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} \equiv \begin{bmatrix} \frac{\partial n_1^{m+1}}{\partial n_1^m} & \frac{\partial n_1^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_1^{m+1}}{\partial n_{S^m}^m} \\ \frac{\partial n_2^{m+1}}{\partial n_1^m} & \frac{\partial n_2^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_2^{m+1}}{\partial n_{S^m}^m} \\ \vdots & \vdots & & \vdots \\ \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_1^m} & \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_{S^m}^m} \end{bmatrix} \tag{12}$$

To find an expression to represent this Jacobian matrix, we arbitrarily select the i, j element of the matrix in accordance with Hagan's derivation:

$$\begin{aligned}\frac{\partial n_i^{m+1}}{\partial \mathbf{n}_j^m} &= \frac{\partial\left(\sum_{l=1}^{S^m} w_{i,l}^{m+1} a_l^m + b_i^{m+1}\right)}{\partial n_j^m} = w_{i,j}^{m+1}\frac{\partial a_j^m}{\partial n_j^m} \\ &= w_{i,j}^{m+1}\frac{\partial f^m(n_j^m}{\partial n_j^m} = w_{i,j}^{m+1}\dot{f}^m(n_j^m),\end{aligned} \tag{13}$$

where

$$\dot{f}^m(n_j^m) = \frac{\partial f^m(n_j^m)}{\partial n_j^m}. \tag{14}$$

Thus, we can write the Jacobian from before as

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \mathbf{W}^{m+1}\dot{\mathbf{F}}^m(\mathbf{n}^m), \tag{15}$$

where

$$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \dots & 0 \\ 0 & \dot{f}^m(n_2^m) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \dot{f}^m(n_{S^m}^m) \end{bmatrix}. \tag{16}$$

Now the recurrence relation for the hidden layer sensitivities can be written as

$$\begin{aligned}\mathbf{s}^m &= \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T\frac{\partial \hat{F}}{\partial \mathbf{n}^{m+1}} \\ &= \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T\mathbf{s}^{m+1},\end{aligned} \tag{17}$$

where $\hat{F}$ denotes the performance index as it is an approximation of the network error [Dem+14]. But, we know the network architecture and performance index used, so we can refine the general equation. The transfer function in the hidden layer is the tangent-sigmoid function and we find the derivative as follows:

$$\begin{aligned}\dot{f}^1 &= \frac{d(\tanh(n))}{dn} = \frac{d}{dn}\left(\frac{\sinh(n)}{\cosh(n)}\right) \\ &= \frac{\cosh^2(n) - \sinh^2(n)}{\cosh^2(n)} = 1 - \tanh^2(n).\end{aligned} \tag{18}$$

Therefore,

$$\dot{\mathbf{F}}^1(\mathbf{n}^1) = \begin{bmatrix} 1-\tanh^2(n_1^1) & 0 & \dots & 0 \\ 0 & 1-\tanh^2(n_2^1) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1-\tanh^2(n_{10}^1) \end{bmatrix}. \tag{19}$$

and the first layer sensitivity is

$$\mathbf{s}^1 = \dot{\mathbf{F}}^1(\mathbf{n}^1)(\mathbf{W}^2)^T\mathbf{s}^2. \tag{20}$$

The only thing left to do is calculate the sensitivity for the output layer, $\mathbf{s}^2$. Calculating the sensitivity for the output layer is a little more difficult as the derivative of the performance index with respect to the net input of the output layer is indirectly a function of the net input of the hidden layer. We can derive the sensitivity for the output layer as follows:

$$\begin{aligned} \mathbf{s}^2 = \frac{\partial \hat{F}}{\partial \mathbf{n}_i^2} &= -\sum_{i=1}^{2} \frac{\partial \mathbf{t}_j \ln \mathbf{a}_j}{\partial \mathbf{n}_i} \\ &= -\sum_{j=1}^{2} \mathbf{t}_j \frac{\partial \ln \mathbf{a}_j}{\partial \mathbf{n}_i} = -\sum_{j=1}^{2} \mathbf{t}_j \frac{1}{\mathbf{a}_j}\frac{\partial \mathbf{a}_j}{\partial \mathbf{n}_i} \end{aligned} \tag{21}$$

where i denotes the i-th neuron and

$$\frac{\partial a_i}{\partial n_i} = \dot{\mathbf{F}}^2. \tag{22}$$

Now we find the derivative of the *softmax* transfer function to be used in the sensitivity calculation:

$$\begin{aligned} \text{if } i = j\text{:}\ \frac{\partial a_i}{\partial n_i} &= \frac{\partial \mathbf{softmax}(n_i)}{\partial n_i} = \frac{e^{n_i}}{\sum_{n=1}^{2} e^{n_i}}\left(\frac{\sum_{n=1}^{2} e^{n_i} - e^{n_i}}{\sum_{n=1}^{2} e^{n_i}}\right) \\ &= \frac{e^{n_i}}{\sum_{n=1}^{2} e^{n_i}}\left(1 - \frac{e^{n_i}}{\sum_{n=1}^{2} e^{n_i}}\right) = a_i(1-a_i) \end{aligned} \tag{23}$$

$$\text{if } i \neq j\text{:}\ \frac{\partial a_i}{\partial n_j} = \frac{\partial \mathbf{softmax}(n_i)}{\partial n_j} = -\frac{e^{n_i}}{\sum_{n=1}^{2} e^{n_i}}\frac{e^{n_j}}{\sum_{n=1}^{2} e^{n_j}} = -a_i a_j.$$

Using this result to finish the derivation of $\mathbf{s}^2$ from (21), we have:

$$\begin{aligned} s_i^2 &= -\sum_{j=1}^{2} \mathbf{t}_j \frac{1}{\mathbf{a}_j} \frac{\partial \mathbf{a}_j}{\partial \mathbf{n}_i} = -\frac{\mathbf{t}_i}{\mathbf{a}_i} \frac{\partial \mathbf{a}_i}{\partial \mathbf{n}_i} - \sum_{j \neq i}^{2} \frac{\mathbf{t}_j}{\mathbf{a}_j} \frac{\partial \mathbf{a}_j}{\partial \mathbf{n}_i} \\ &= -\frac{\mathbf{t}_i}{\mathbf{a}_i} \mathbf{a}_i (1 - \mathbf{a}_i) - \sum_{j \neq i}^{2} \frac{\mathbf{t}_j}{\mathbf{a}_j} (-a_i a_j) \\ &= -\mathbf{t}_i + \mathbf{t}_i \mathbf{a}_i + \sum_{j \neq i}^{2} \mathbf{t}_j \mathbf{a}_i = -\mathbf{t}_i + \sum_{j=1}^{2} \mathbf{t}_j \mathbf{a}_i \\ &= -\mathbf{t}_i + \mathbf{a}_i \sum_{j=1}^{2} \mathbf{t}_j = \mathbf{a}_i - \mathbf{t}_i. \end{aligned} \tag{24}$$

Now we have the starting sensitivity:

$$\mathbf{s}^2 = \mathbf{a} - \mathbf{t} \tag{25}$$

and have everything we need to backpropagate the sensitivities for our network.

4.2.4 Update Rule

Lastly, we need to update the weights and biases via an update rule. For the replication in Python, we use the approximate steepest descent rule to update weights and biases:

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T, \tag{26}$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha \mathbf{s}^m. \tag{27}$$

5 Experimental Setup

This section will document how the data are used to train and test the network, implementation of the model in Python, and performance metrics used to evaluate the final model.

5.1 Data Preprocessing

The data were almost ready to be used when we downloaded it as there were no missing values. The only thing that needed to be changed from the original dataset was the target values. Originally, these values were coded on a one-to-seven scale where values of one-to-four denoted a type of window glass and values of five-to-seven denoted non-window glass. When coding the targets for pattern recognition problems, it is best practice to have as many dimensions in the targets as there are classes. This ensures only one neuron

is active at a time and generally produces better results. Further, we coded the targets as $\begin{bmatrix}1\\0\end{bmatrix}$ if the glass sample comes from a window and $\begin{bmatrix}0\\1\end{bmatrix}$ it it does not for this reason. We briefly mention this in Section 3.2 during our discussion of the targets. The code used to import the data and code the targets can be found in Appendix A.

After collecting the data, we normalize it based on the network architecture we have chosen. Since we use a tangent-sigmoid transfer function in the hidden layer, it is very important that we normalize the inputs between negative one and one since this is the range of the transfer function. When the net input is large (or small), the function becomes saturated. We do not want this to happen at the beginning of the training process because the gradient will be close to zero. Since the first layer's net input is the product of the weight and input plus the bias, large inputs can result in large net inputs if the weights are not small enough scale the inputs down. Thus, it is important to normalize the inputs before we feed them to the network. In order to normalize the inputs to values between negative one and one, we use the following equation:

$$\mathbf{p}^n = 2(\mathbf{p} - \mathbf{p}^{min})./(\mathbf{p}^{max} - \mathbf{p}^{min}) - 1, \tag{28}$$

where $\mathbf{p}^{min}$ is the vector containing the minimum values of each element of the input vectors in the dataset, $\mathbf{p}^{max}$ contains the maximum values, ./ represents the element-wise division of the vectors, and $\mathbf{p}^n$ is the normalized input vector [Dem+14].

Next, we divide the data up into three different sets—training, validation, and testing. We chose to use 70% of the data for training, 15% of the data for validation, and 15% of the data for testing. This is fairly standard practice and follows the literature. To ensure that each of these sets is representative of the full dataset, we split the the data up into two subsets—input-target pairs for window samples and input-target pairs for non-windows.

Then, we split each of these subsets up into training, validation, and testing sets using the 70:15:15 ratio. After splitting the subsets up into training, validation, and testing, we combine each split-up subset accordingly—i.e. window-training subset with non-window-training subset. This gives us the highest chance of having our subsets be representative of the full dataset and that the subsets cover the same region of the input space. Once the subsets have been combined into complete training, validation, and testing sets, we reshuffle the input-target pairs so that they are randomly-ordered again. The subroutine used to normalize the inputs and split the data can be found in Appendix B.

5.2 Implementation of the Network

To implement the network, we created a Python class, `NeuronLayer`, to represent layers of a neural network. The `NeuronLayer` class takes the dimension of the inputs, number of

neurons in the layer, transfer function, and gradient of the transfer function as arguments. Moreover, there are several functions defined to help train and implement the network in Python. The subroutine used to create the `NeuronLayer` class can be found in Appendix C.

We begin by instantiating layers from the created class to mimic the network architecture we designed. We arbitrarily chose a learning rate, α, of 0.01. If the learning rate converges too slowly, then we will increase the learning rate accordingly. Then, we randomly initialize the weights and biases on the interval $[-0.5, 0.5]$ and pass these on to the corresponding network layers with the `setWeightBias` method. Weights and biases are usually initialized on this interval when we have normalized our inputs between negative one and one. Lastly, we initialize empty vectors to hold the error from each training iteration. These steps can be found in Appendix D.

5.3 Training the Network

We chose to train the network over 100 epochs where one epoch represents a single batch implementation of approximate steepest descent. The training process begins by propagating each input in the training set forward by calling the `FP` method of the `NeuronLayer` class. Then, the cross-entropy is calculated using the equation found in (8). This error is then used to calculate a sensitivity and then propagated backward through the model. Once this process is done for each input in the dataset, we approximate the cross-entropy loss of the epoch by taking the mean of the cross-entropy of the given epoch. Then, we use this information in addition to the learning rate to update the weights and biases by (26) and (27), respectively.

In attempts to speed up the iterative process used to train the model, we perform the validation simultaneously. We use the same process to propagate inputs forward and compute the cross-entropy loss, but instead of updating the weights and biases based on the validation set, we use the weights and biases from the training set. This allows us to see if the network is training properly since the validation set is information our model has not been trained on. If the error keeps decreasing in the validation set as the number of iterations increases, this means our network is training properly. To ensure we do not overfit the data with our model, we implement an early stopping technique. We decided to have our network stop training if the validation error increased for five *consecutive* epochs. If this happens to be the case, the script for the model will print to the console that training was stopped early and at what iteration it stopped. The code used to implement this can be found in Appendix E or in the include file `NeuronLayer.py`.

6 Results

7 Conclusion

References

[ES87] Ian W Evett and EJ Spiehler. "Rule Induction In Forensic Science". In: *KBS in Goverment, Online Publications* (1987), pp. 107–118.

[MA94] P.M. Murphy and D.W Aha. "UCI Repository of Machine learning Databases [http://www.ics.uci.edu/ mlearn/MLRepository.html], Department of Information and Computer Science". In: *University of California, Irvine, CA* (1994).

[Bot09] Maureen C Bottrell. "Forensic Glass Comparison: Background Information Used in Data Interpretation". In: *Forensic Science Communications* 11.2 (2009).

[AT12] Cahoon Almirall Naes and Trejos. "Elemental Analysis of Glass by SEM-EDS, μXRF, LIBS and LA-ICP-MS". In: *National Criminal Justice Reference Series* (2012).

[Dem+14] Howard B. Demuth et al. *Neural Network Design*. 2nd. USA: Martin Hagan, 2014. ISBN: 0971732116, 9780971732117.

A Importing the Data & Coding the Targets

The Python code used to import the data and code the targets is listed below. First the data are imported with a standard `read_csv` statement from the `pandas` library. Then, the targets are coded by the process stated in Section 3.2. The full subroutine can be found in the included file `DataPrep.py`.

```
def GlassImport():
    # import dataset
    cols = ['id', 'RI', 'Na', 'Mg', 'Al', 'Si', 'K', 'Ca', '
   Ba', 'Fe', 'type']
    glass = pd.read_csv('glass.txt', names=cols)
    # create window target column (1 for true; 0 for false)
    glass.ix[glass.ix[:, 'type'] <= 4, 'window'] = 1
    glass.ix[glass.ix[:, 'type'] > 4, 'window'] = 0
    # create non-window target column (1 for true; 0 for
   false)
    glass.ix[glass.ix[:, 'type'] > 4, 'non_window'] = 1
    glass.ix[glass.ix[:, 'type'] <= 4, 'non_window'] = 0
    # drop id and class columns
    glass.drop(['id', 'type'], inplace=True, axis=1)
    return glass
```

B Data Normalization & Division

The following code documents the process used to normalize the inputs and then divide the data up into training, validation, and testing sets. The full subroutine can be found in the included file `Split.py`.

```
def split(dataset, num_inputs, t, v):
    # normalize dataset into range between -1 and 1
    norm = dataset.ix[:, slice(0,num_inputs)].apply(lambda x:
    -1 + 2 * (x - x.min()) / (x.max() - x.min()), axis=0)
    dataset = pd.concat([norm, dataset.ix[:,'window':]], axis
   =1)
    # split dataset by target class
    dataset0 = dataset.ix[dataset.ix[:, 'window'] == 0, :]
    dataset1 = dataset.ix[dataset.ix[:, 'window'] == 1, :]
    # build datasets by target class
    train0, validate0, test0 = np.split(dataset0.sample(frac
   =1),
                                        [int(t * len(dataset0
   )), int((1 - v) * len(dataset0))])
    train1, validate1, test1 = np.split(dataset1.sample(frac
   =1),
                                        [int(t * len(dataset1
   )), int((1 - v) * len(dataset1))])
    # re-combine by target class
    train = pd.concat([train0, train1])
    validate = pd.concat([validate0, validate1])
    test = pd.concat([test0, test1])
    # shuffle order and reset index numbers
    train = train.sample(frac=1)
    train.reset_index(inplace=True)
    validate = validate.sample(frac=1)
    validate.reset_index(inplace=True)
    test = test.sample(frac=1)
    test.reset_index(inplace=True)
    return train.iloc[:, 1:], validate.iloc[:, 1:], test.iloc
   [:, 1:]
```

C Python Class for Network Layers

The following code shows the creation of the `NeuronLayer` class. The full subroutine can be found in the included file `NLObjects.py`.

```
class NeuronLayer:
    # define attributes
    def __init__(self, r, s, f, j):
        self.r = r # Number of inputs
        self.s = s # number of neurons
        self.f = f # Transfer function
        self.j = j # derivatve transfer function

    def setWeightBias(self, w, b):
        self.w = w  # set weights
        self.b = b # set biases
        self.r = w.shape[1]
        self.s = w.shape[0]
        return

    def FP(self, p):
        self.p = p
        self.n = self.w * self.p + self.b
        self.a = self.f(self.n)
        return

    def update(self, learning_rate, sensitivity):
        self.w = self.w - learning_rate * sensitivity * self.
    p.T
        self.b = self.b - learning_rate * sensitivity
        return
```

D Initializing the Network Architecture in Python

This code shows the process used to instantiate layers and initialize the parameters of the network.

```
# ---------------------- specify network architecture
    ----------------------
num_neurons1 = 10  # layer 1
num_neurons2 = 2   # layer 2
alpha = 0.01       # learning rate
epoch = 100        # number of iterations
nlayer1 = nl.NeuronLayer(num_neurons1, num_inputs, nl.tansig,
    nl.j_tansig)      # instantiate layer 1
nlayer2 = nl.NeuronLayer(num_neurons2, num_neurons1, nl.
    softmax, nl.j_softmax)  # instantiate layer 2
np.random.seed(0)
# randomly initialize weight and bias on the interval [-0.5,
    0.5]
W1 = np.matrix(np.random.rand(num_neurons1, num_inputs) -
    0.5)
b1 = np.matrix(np.random.rand(num_neurons1, 1) - 0.5)
W2 = np.matrix(np.random.rand(num_neurons2, num_neurons1) -
    0.5)
b2 = np.matrix(np.random.rand(num_neurons2, 1) - 0.5)
# initialize cross entropy loss (training & validation)
ce_t = []
ce_v = []
global s1_all, s2_all
# pass on randomly initialized weights and biases to the
    network
nlayer1.setWeightBias(w=W1, b=b1)
nlayer2.setWeightBias(w=W2, b=b2)
```

E Early Stopping

The code below shows the early stopping condition used to prevent the model from overfitting.

```
if j == 0:
 val_fail = []
elif ce_v[j] > ce_v[j-1]:
 val_fail.append(1)
 if len(val_fail) == 5:
  print 'Validation error has increased for 5 consecutive
   epochs. Early stopping at epoch {}'.format(j)
  break
else:
 val_fail = []
```