Capstone Project

Stewart DeSoto

Udacity Machine Learning Nanodegree

July 28, 2016

Definition

Project Overview

Mapping the internal state of the brain to observable physical effects or thought patterns is one of the greatest topics of interest in the fields of neuropsychology and Artificial Intelligence today. For this project, I will examine EEG (electro-encephalogram) signals observed in a subject's brain over a period of 117 seconds. During this period, the subject repeatedly opened and closed his eyes, and these external state transitions were recorded by an observer. The goal of the project is to use 14 numerical EEG signals to predict the subject's binary eye state (open/closed) at any given time.

A highlight of this project is that I plan to implement, from scratch, and without referring to any neural network design references during this project, my own Artificial Neural Network (ANN) in order to make the desired eye state classifications.

Problem Statement

In order to implement the classifier, we will download the data from the University of California, Irving Machine Learning Data Set[1]. The goal is to create a binary classifier to accurately predict eye state given brain signals, which consists of approximately 15,000 observations of the EEG measurements, which consist of 14 decimal values.

Metrics

The simplest metric for binary classification is the accuracy score of the predictions, defined as:

$$Accuracy = \frac{Number\ of\ Correct\ Predictions}{Total\ Number\ of\ Predictions}$$

[1] "EEG Eye State Data ",https://archive.ics.uci.edu/ml/datasets/EEG+Eye+State,

It is interesting to note that a classifier with no predictive power would have an accuracy of 50% (or 0.5), meaning that its predictions are no better than flipping a coin. If the accuracy was 0.0, then the opposite class could always be predicted, and the accuracy would actually be 100% (1.0).

The simple metric of classification accuracy is often not the most helpful indicator of the performance of a classifier, especially for data sets that are imbalanced (many more negative than positive examples). For this reason, we will also use a more sophisticated evaluation metric for comparison. This is the Receiver Operating Characteristic (ROC) curve, often used in machine learning competitions to grade binary classifiers. Here, the True Positive Rate (TPR) is plotted vs. the False Positive Rate (TPR). These values are calculated for a given set of true labels and predicted probabilities by moving the decision boundary threshold up from 0 to 1 and calculating TPR and FPR at each threshold. In this case, the figure of merit is the Area Under Curve (AUC). A perfect classifier would have an AUC value of 1.0, while a random classifier has an AUC equal to 0.5

It is important to note that these metrics must be measured using a test/validation set, which is disjoint from the training data set used to determine the parameters of the model; in our case, these are the neural network weights. It is easy to overfit your model and obtain performance results that do not generalize well to new examples if you use training data to characterize model performance.

Analysis

Data Exploration

As discussed earlier, each example consists of 14 measurements of the electrical activity in certain regions of the brain over each short time interval. There are 14,980 examples taken over a period of 117 seconds, so each sample characterizes activity over a period of 7.8 ms. The mean and standard deviation of each of the 14 features is provided in this table[2]:

Feature	1	2	3	4	5	6	7	8
Mean	4322	4010	4264	4165	4342	4644	4110	4616
Std Dev	2492	46	44	5216	35	2924	4600	29

Feature	9	10	11	12	13	14
Mean	4219	4231	4202	4279	4615	4416
Std Dev	2136	38	38	41	1208	5891

[2] See EEG_Eye_exploration.ipynb for code.

It is apparent that the means of the EEG readings are clustered in the narrow range from 4000 to 4700. The majority of the standard deviations are close to 40, but several are much, much larger, reaching into the 4000-5000 range. These wide distributions must result from extreme outliers, perhaps errors in the EEG sensors at startup, or if the subject moved his head, etc. Let's examine the distributions of the first two features in more detail:

Feature	1	2
Count	14980	14980
Mean	4322	4010
Std	2492	46
Min	1034	2830
25%	4281	3991
50%	4294	4006
75%	4312	4023
max	309231	7805

Here we can clearly see the presence of outliers in feature 1, with the maximum measurement a staggering 122 standard deviations above the mean! We may need to exercise care when fitting our model because very large features can have outsize influence on weight updates in the back propagation algorithm.

The following table summarizes the distribution of class labels in our dataset. The output classes are fairly well balanced.

Eye State	Label	Count
Open	0	8257
Closed	1	6723

Exploratory Visualization

In the following figure, we plot a histogram of four randomly selected features, including only those points within the main distributions (and thus ignoring outliers). The features appears to be fairly normally distributed, although there is positive skew for feature 1 and multiple peaks for feature 3.

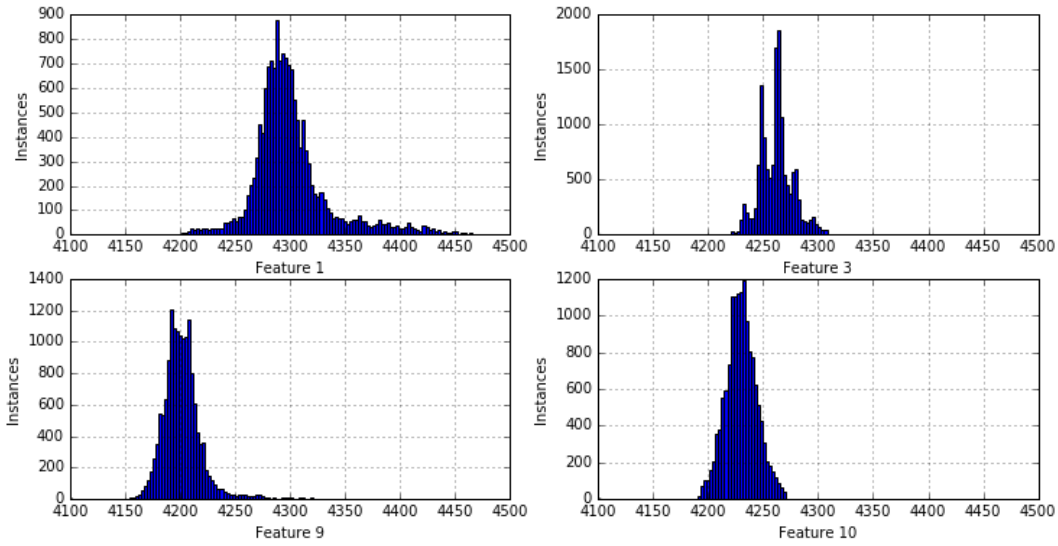

Fig. 1. Distributions of four of the 14 input feaures.

It is also illustrative to graphically show the opening and closing of the subject's eyes during the 117-second duration of the experiment as labeled by the experimenter. In the following figure, the blue ranges indicate the eyes' closed state. The subject appears to have closed their eyes more in the first half of the time window, with mostly brief blinks in the second half.

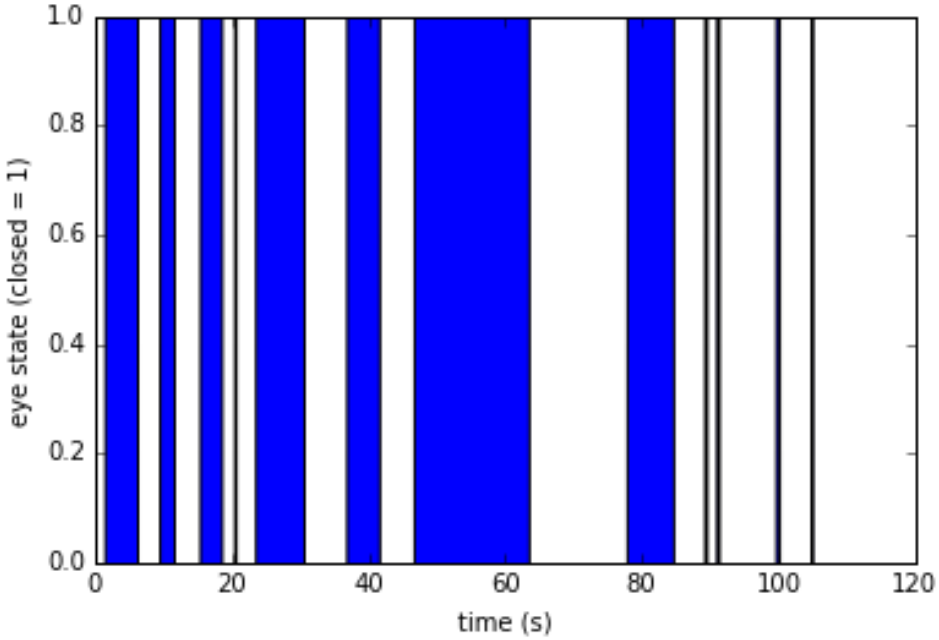

Fig. 2 Plot showing the eye-state over the 117 second trial.

Algorithms and Techniques

There are many supervised learning approaches for performing binary classification for a set of input examples, with a group of feature values used to predict a binary output label. Some of the more popular (and elementary) algorithms include logistic regression and decision tree classifiers. More sophisticated techniques include ensemble methods such as Boosted Trees and Random Forests and many other methods (for example, Support Vector Machines, and Artificial Neural Networks).

For this project, I decided to implement my own neural network, from scratch, deriving all of the mathematical equations necessary to carry out the feed-forward prediction steps, and also the backpropagation algorithm, which updates the weights connecting the adjacent network layers. This design is also known as a multi-layer perceptron with backpropagation. Since this is my first time developing a neural network, and I will reproduce all of the math on my own, I will limit the architecture to a single hidden layer, which may have a variable number of hidden units. The inputs to the neural network will be the 14 decimal EEG values. In addition, bias units, which always contain "1's", are added to the input and hidden layers. These are needed to move the decision boundary hyperplane away from the origin.

Neural networks, especially in the form of deep networks composed of multiple layers, and with bidirectional Restricted Boltzmann Machines (RBMs), are capable of learning complex, underlying features, and are therefore prone to overfitting. It is our expectation that our data set (~15,000 examples) and simplified neural network architecture (only one hidden layer) will generalize well to test data.

Benchmark

As discussed earlier, we hope to implement several metrics commonly used in binary classification. For the simplest, classification accuracy, we hope to achieve an accuracy score of 80%, well above the level one would expect for a random classifier (50 %). We will also measure the Area Under Curve of the ROC curve and expect to see at least 0.80 for this metric.

Methodology

Data Preprocessing

There is minimal processing of the provided data set before we can feed it to our neural network. This is one of the best qualities of neural networks, and especially deep networks, in that minimal manual feature selection and feature extraction need to be performed. The large numerical values see in the input data (4000-5000) indicated that initial weightings with a unit standard deviation around zero may

take a while to learn. For this reason, we will normalize the 14 features to have zero mean and unit variance. This is done by performing the following calculation on each of the 14 EEG features, separately:

$$Scaled\ Feature_{i,j} = \frac{Raw\ Feature_{i,j} - Mean\ Feature_i}{Standard\ Deviation_i}$$

where $1 \leq i \leq 14$ is the feature number, and $1 \leq j \leq N$ is the example number. We use the StandardScaler module in sklearn for this.

Implementation

The newest beta version of Scikit-Learn has a neural network model, but previous versions do not. There are other neural networks libraries available in Python, such as PyBrain[3], but I thought it would be appropriate for this capstone project to go a different route and build my own. Since I am building my own neural network model from scratch, the implementation must begin with coding the model in Python. The neural network is formulated as a Python class[4] . The architecture for my neural network is shown below.

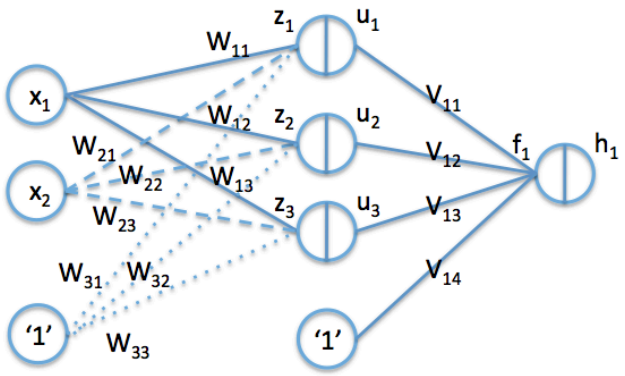

Fig. 3 The Neural Network topography showing variables used in model.

Note that the model allows for a variable number of input features and hidden units, although only for a single hidden layer.

Example equations for the forward propagation path are as follows:

$z_1 = W_{11}\, x_1 + W_{12}\, x_2 + W_{13} * 1$ (linear combination of inputs)

---

[3] http://pybrain.org/ "PyBrain" stands for **Py**thon-**B**ased **R**einforcement Learning, **A**rtificial **I**ntelligence and **N**eural Network Library

[4] code in `neuralnetwork.py` and `run_nn_eeg.py`

$u_1 = \frac{1}{1+e^{-z_1}}$ (the sigmoid activation function)

$f_1 = V_{11}\, u_1 + V_{12}\, u_2 + V_{13} * 1$ (linear combination of hidden units)

$h_1 = \frac{1}{1+e^{-f_1}}$ (another sigmoid activation function)

In the Python implementation, these equations are turned into matrix form for efficient calculation.

The weight updates are done via backpropagation, which is form of gradient descent. Thus, we need to find the gradient of the model's error term with respect to each of the weights in the matrices *W* and *V*. We will assume that the classification error is given by the sum of squared errors (where $y_i \in (0,1)$ is the labeled output) and $0 \leq h_i \leq 1$ is the neural network output predicted probability for output *i*.

$$E = \sum_{output\ i} (y_i - h_i)^2$$

The gradient term is found via the chain rule, starting at the error term, and following the links back to the weight we wish to update. For example, the update of the W_{12} can be found from:

$$\frac{\partial E}{\partial W_{12}} = \frac{\partial E}{\partial h_1}\frac{\partial h_1}{\partial f_1}\frac{\partial f_1}{\partial u_1}\frac{\partial u_1}{\partial z_1}\frac{\partial z_1}{\partial W_{12}}$$

If we recall that the derivative of the sigmoid function can be written in terms of the sigmoid itself, we can easily work out the partial derivatives to obtain

$$\frac{\partial E}{\partial W_{12}} = (y_1 - h_1) \quad h_1(1 - h_1) \quad V_{11} \quad u_1(1 - u_1) \quad x_2$$

Thus, we see that there will only be a nonzero update to a weight if the predicted label output doesn't match the actual label. Also note that large inputs x_i have a large impact on the gradient.

We will have many similar equations to compute, one for each weight, so we write all of these derivatives in an efficient matrix form. To complete the backpropagation updates, we then simply multiply the gradient vector by a small learning rate numerical value, and incrementally update the weights throughout the network.

Even though this particular brain modeling application doesn't require multiple outputs from the network, I included the functionality for future use. I also modeled the NeuralNetwork class's API (Application Programming Interface) after the Scikit-Learn machine-learning library. In particular, a neural network object can be

instantiated with the parameters desired, including the number of hidden units, learning rate, number of iterations, etc.

After the neural network is created, a *NeuralNetwork.fit()* instance method can be called. This method takes in the features as a matrix, with each row representing a certain EEG sample. Our NN model assumes a supervised learning environment, in which the labels (eye state open/closed) corresponding to each EEG sample are also provided to the method.

Inside the fit method, the model shuffles all of the examples over each pass through the training set. Then, for each EEG/eye state pair, the neural network calculates the output activation, and performs back-propagation to update the weights from the inputs to the hidden units (the matrix *W* in our code), and also the weights from the hidden units to the outputs (matrix *V* in our code).

The squared error between each label and the neural network output is accumulated for each feed forward calculation, so that training error as a function of training epoch can be followed.

Finally, *NeuralNetwork.predict()* and *NeuralNetwork.predict_prob()* instance methods can be called to calculate any required classification metrics.

Refinement

In the initial implementation of the neural network, there was no inclusion of bias units. Such a model can work well for some datasets, but not for the general case (i.e., for sample spaces which don't have decision boundary which pass through the origin.).

Also, in the first learning pass I ran on the EEG dataset, I used the default number of hidden units (set at 2), which, as we will see, is insufficient to capture the complexity inherent in the EEG data. Subsequently, I increased the number of hidden units to 8, and found a more rapid decrease of the Sum of Squared Errors (SSE) over each pass of the 15,000 samples in the dataset and better classification accuracy.

Results

Model Evaluation and Validation

The neural network was run twice, with a varying number of hidden units and learning rate. For training, we used 75% of the available EEG samples, and reserved the remaining 25% of the labeled data for estimating our out-of-sample accuracy.

The graph below shows the squared error for each iteration during the learning phase for three different learning runs on the dataset.

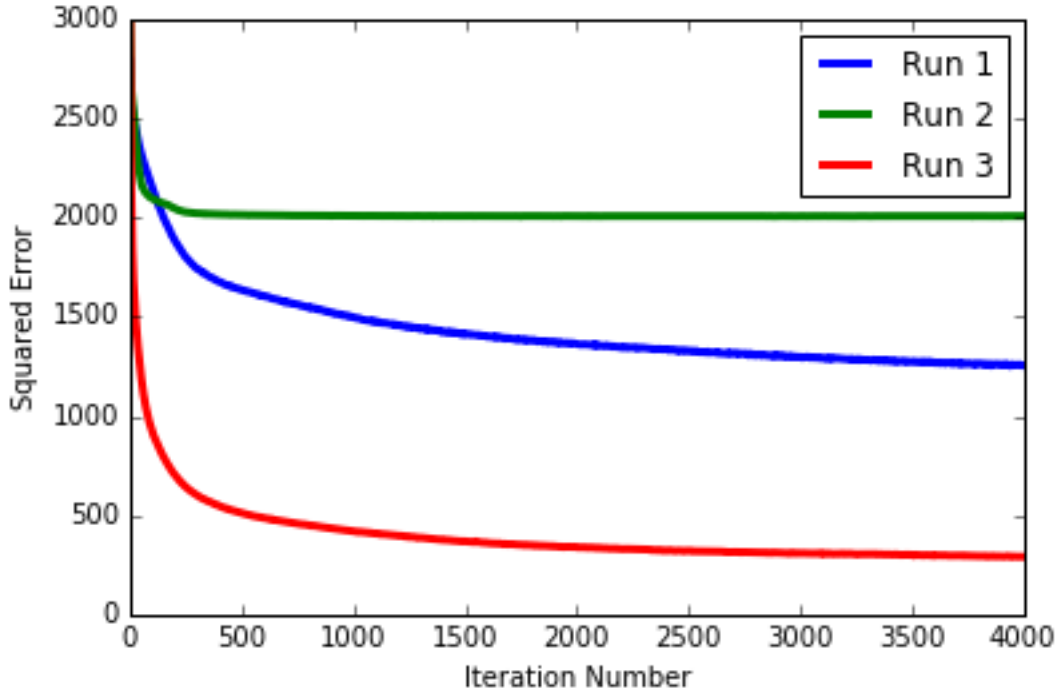

Fig. 4 The model training curve for runs 1 to 3.

It is clear that the learning is happening in our neural network model, since the squared error is decreasing as the backpropagation algorithm slowly adjusts the weights, which then more accurately predict the output eye-state labels. For run 1, we use all of the 14,980 examples and 24 hidden units. For run 2, we drop the 3 extreme outlier examples, and also restrict the hidden layer to just 2 hidden units. Finally, for run 3, we again use 24 hidden units, and also drop the extreme outliers.

The classification results for the two models are summarized in the table below [5]:

Run	Num Hidden Units	N_iter	Examples Dropped (of 14980)	Learning Rate	Training Accuracy	Test Accuracy
1	24	4,000	0	0.01	0.846	0.844
2	2	4,000	3	0.01	0.746	0.736
3	24	4,000	3	0.01	0.971	0.940

[5] See `nn_eeg_results.ipynb` and `cross_validation.py` for code, and `nn_eye_eeg._runs.pdf` for output text.

From the graphs and table, it is clear that the number of hidden units strongly impacts the capacity of the neural network and limits its ability to model the EEG data. Also, it is striking that dropping just 3 bad data points out of almost 15,000 can have a dramatic effect on the usefulness of a neural network classifier. This is not surprising, since the gradient update value of the inputs is directly proportional to the input vector x.

There is reasonably good predictive power in our neural network model, as we are able to correctly classify 94% of the eye open/closed state labels in our out-of-sample test set. Furthermore, there is only a hint of overfitting the model, as the test accuracy is only slightly lower than the in-sample training error. This might not be surprising in view of the bias-variance tradeoff in that the current implementation of our neural network has somewhat limited capacity to model features in the input data, given that we are using a single hidden layer.

The ROC curves and AUC measurements for the test data for runs 2 and 3 are shown next.

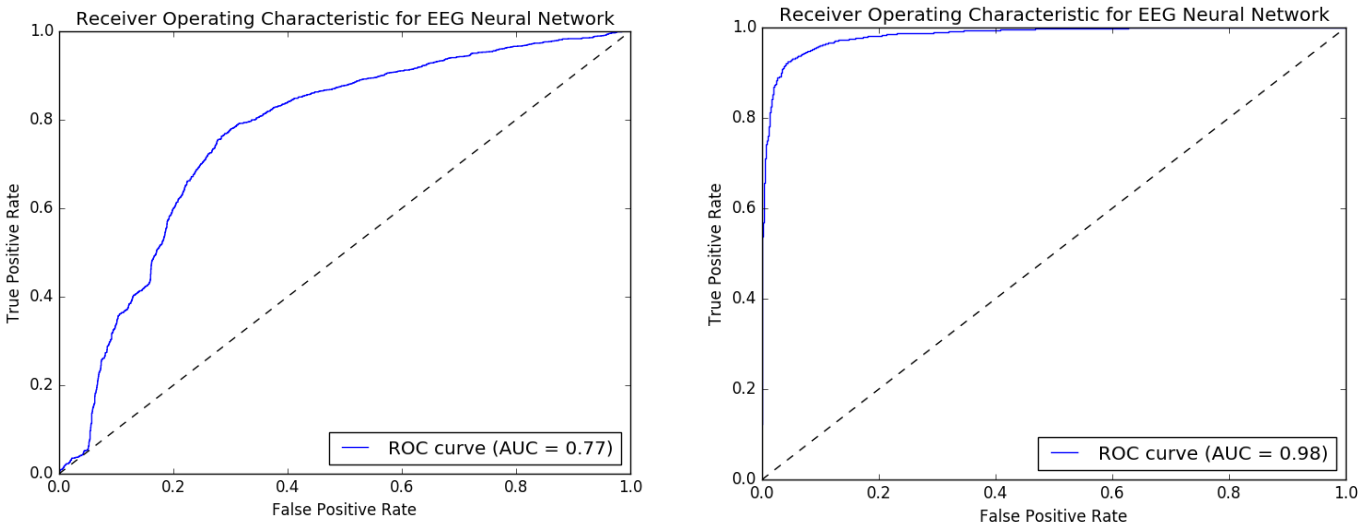

Fig. 5 ROC curve for single hidden layer neural network with num_hidden_units=2 (on left) and num_hidden_units=24 (on right).

The classifier with 24 hidden units clearly outperforms the one with just 2 hidden units.

We could further achieve even better performance by increasing the capacity of our model by adding one or more hidden layers, pushing us toward the “variance” end of the bias-variance tradeoff spectrum. In this case, it would be easier to overfit our

model, and we would need to take additional steps to guard against this, such as L1 and L2 regularization[6] and Dropout[7].

Justification

Our current neural network is clearly performing well. The test data accuracy is only slightly below the training set accuracy, indicating that we can expect the model to perform at a similar level on new, "real world" EEG data.

It's questionable, though, if the final accuracy of my model (94%) is "good enough." In the present scenario, we are only trying to determine the eye state to better model the eye-brain connection, and there are no major safety or other serious issues at stake. One could envision a system monitoring the eyes and brain activity of a driver and alerting them if they begin to drift off to sleep. In that case, an accuracy well above 99% would likely be required.

To get a sense of how well my neural net classifier compares with a typical state-of-the-art, widely used classifier, I fed the same EEG dataset into the RandomForestClassifier from sklearn (code can be found in *EEG_randomforest.ipynb*). I used all of the "out of the box" settings. One immediately obvious difference was the speed of learning the data. My neural net takes about 1 hour to carry out 4,000 iterations over the 15,000 examples in the data set. In contrast, the RandomForestClassifier was done within 2 seconds!

However, my neural network model perform significantly better in test accuracy, and showed much less evidence of overfitting the model to the noise present in the training dataset.

Classifier	Learning Time	Training Accuracy	Test Accuracy
Our NN (24 hidden units)	1 hour	0.971	0.940
Random Forest	2 seconds	0.996	0.885

[6] http://www.kdnuggets.com/2015/04/preventing-overfitting-neural-networks.html/2

[7] "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", Srivastava, et. al., Journal of Machine Learning Research 15 (2014) 1929-1958

Conclusion

Free-Form Visualization

It is interesting to take a step back and think about what is actually happening when a subject opens and closes her eyes. The photons reflecting from the objects that the individually is observing are absorbed in the retina, and in the process transfer their energy to electrons in the optic nerve. These electronic signals travel along various nerves, through the optical chiasm, and into the optical cortex. This process creates the electrical voltages which the EEG mounted on top of the head is able to pick up. I find it intriguing that the exact same neural network model and process in the organic matter of our brains is what we are essentially trying to reverse engineer with our simplistic miniature brain in silicon. This network is visualized in the following figure[8]:

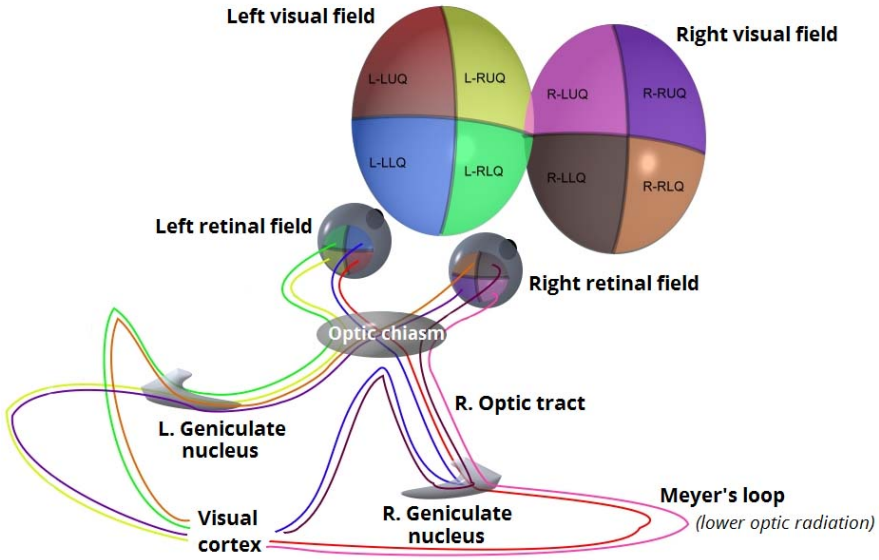

Fig. 7 Electrical connections between the eyes and brain

For one final visualization, it is interesting to examine the trained final matrix of coefficients representing the strength of the neural network coefficients from the 14 EEG inputs to the 24 hidden units in our final neural network model. The graph below is a heat map representation of these coefficients[9]. It is clear, from the large sea of green, that many of the weights have very small magnitudes, while a few dominate the network. It appears that the input features on the right of the map tends to have more significant weights, some positive and some negative It would be interesting to know to which region of the brain these features are mapped to.

---

[8] Obtained from http://teachmeanatomy.info/head/cranial-nerves/optic-cnii/

[9] see `Wmatrix_heatmap.ipynb`

Conversely, input features 4 to 9 have smaller weights connecting them to the hidden units. Perhaps these electrodes are not near optical nerves.

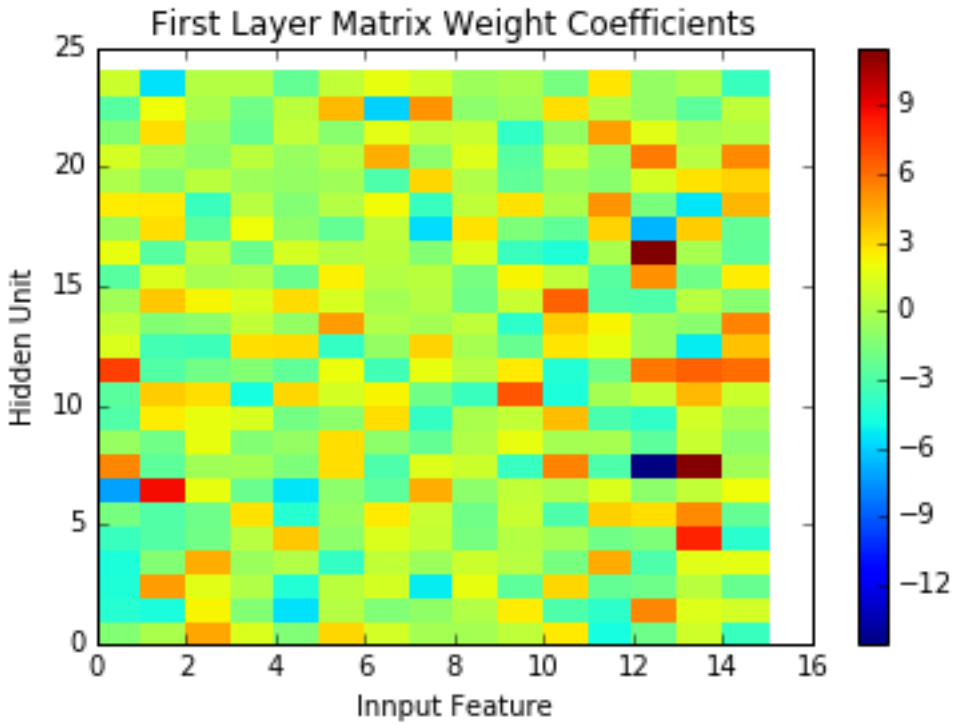

Fig. 8 Heat Map showing the values of coefficients in the first neural network layer for a network with 24 hidden units.

Reflection

This was a great exercise in imbibing the concepts of machine learning at a deep level. It was fun to derive and code a neural network using only a pen and a sheet of paper, and code editor. Figuring out how to write the backpropagation equations in terms of matrix operations was great, but took a little while to do. The matrix outer product was useful in this process of multiplying the various contributing terms in the chain rule. Also, I debated for a while about the best way to add bias units for the input and hidden layers. Initially, I thought it would best have explicit bias variables (e.g., b1 and b2) and to not add actual "1's" to the inputs and hidden layer outputs. In the end, I decided the code was more streamlined and made more logical sense if I added no separate bias weight variables, but instead added bias units to the input vector x, and the hidden unit vector u, and increased the size of the W and V matrices connecting input to hidden layer, and hidden layer to output layer. This allowed the backpropagation routine to work with no change required.

It is a very good lesson to see what a severe detrimental effect bad data can have on the performance of a machine learning algorithm. Who would think that just 3 outlier examples could impact the gradient descent so much?

If time allowed, I would like to do a cross-validated grid search and find the optimal values for the learning rate and number of hidden units, and also study the effect of adding additional hidden layers. I assume that there is a theoretical basis for choosing the optimal neural network architecture (number of layers, and number of hidden units in each layer), but I suppose that in practice these hyperparameters are best determined by using such techniques as grid search and cross validation.

Improvement

There are obviously many ways in which our neural network classifier could be improved.

In terms of improving computational efficiency, likely its greatest need, our code currently doesn't take advantage of the fact that many of the terms for backpropagation are already computed during the feed forward calculation. Similarly, during the backpropagation steps, results from previous layers can be used in subsequent layers as we work our way through the chain of derivatives. It would be very interesting to quantitatively measure the time gains by making these and other changes. I would be very interested to see what other backpropagation routines do to speed up learning.

Beyond this, the most glaring improvement needed is to increase the capacity of the neural network architecture. Being limited to a single hidden layer is quite restrictive. It would not be too difficult to create a single numpy array that would contain all of the weights for all of the layers, so that our current W and V matrices, and those for any additional layers would all be included in this 3-dimensional array. I have heard that it is difficult to train deeper neural networks, and that issues such as the "vanishing gradient" problem result in tiny incremental changes to the layer weights for layers near the inputs. It would be fun to tackle this problem and to see what has already been learned by humans to make machine learning in neural networks more efficient.