CS181 Assignment 2: Neural Networks

Ina (Weilin) Chen and Kathy Lin

February 25th, 2013

1. (a) Not possible. This is the XOR problem which is not linearly separable and thus is a feature that cannot be recognized by a perceptron. We can prove that no such perceptron exists by setting the value of each pixel to be 0 if off and 1 if on and give all of them the weight of $\frac{1}{9}$ (it won't make sense to have different weights as we do not know what region would be on or off). In this case, the perceptron should return 1 if the sum is greater than $\frac{3}{4}$ or less than $\frac{1}{4}$, which is impossible for a Boolean function.

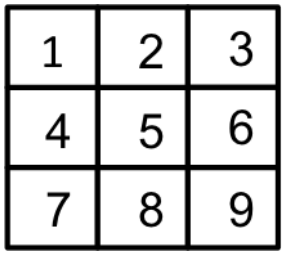
   (b) Possible. One possible perceptron would have weight vector

   $$x = \{\frac{1}{3}, \frac{1}{3}, \frac{1}{3}, -\frac{1}{6}, -\frac{1}{6}, -\frac{1}{6}, -\frac{1}{6}, -\frac{1}{6}, -\frac{1}{6}\}$$

   and

   $$g(s) = \begin{cases} 1, \text{ if } s > 0 \\ -1, \text{ else} \end{cases}$$

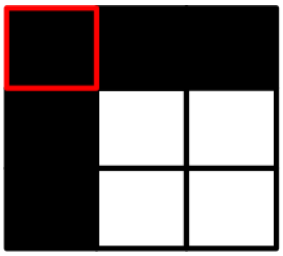
   where x_i is the weight of the corresponding square pictured below

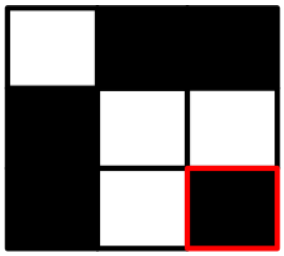
   

   (c) Not possible. This is the connectedness problem which also cannot be classified by perceptrons. The perceptron only look at local features and thus cannot zoom out to see connectedness on the whole.

   We can prove this with the following example. The weights on each of the squares must equal by symmetry where the same configuration

would still be connected if we rotated or flipped the configuration. Now consider a configuration like the one on the left below, the perceptron would label it as connected. However, if one of the squares was moved to another part of the grid (below, right), the perceptron would have to return the same label but the configuration is no longer correct. Thus there is no perceptron that is able to detected if all the lit up blocks were connected.





2.

- Decision Trees
  Con: Would not be able to take into account relation of pixels to one another. This an important function to have as analysis of image would involve looking at how the pixels are related rather than the specific locations of the image.
  Pro: Faster running time (because it doesn't have to look at every feature, reducing the amount of things the algorithm has to run).
- Perceptrons
  Con: Requires linear separability which may be hard to get with complex inputs like images. Moreover, perceptrons would not be able be flexible about the location of the digit on the image. (We might have weights in the shape of the digit we're looking for but if the digit in the image is off-center, for example, it would mess up the weighing).
  Pro: Allows features to be combined (though only once). This is good as images rely on spatial relations but with just one perceptron this is still limited.
- Boosted stumps
  Con: Not good with noisy data (which we have here with mislabeling, different handwritting styles, and digit separation errors)
  Pro: Won't overfit.
- Multilayer Feed-Forward Neural Network
  Con: Slow convergence (especially for such a large data set)
  Pro: Can have many levels of feature extractions, especially the high-level feature extractions, which would be very useful for generalizing the relations between pixels in images.

We would choose the multilayer feed-forward neural network for the task of recognizing handwritten digits. The multilayer neural network's property of feature extraction would be extremely powerful for this purpose while the shortcomings of other methods (such as sensitivity to noise, need for linear separability, and lack of feature relation analysis) make the other methods much less attractive.

3. Problem 5:
   We want to normalize the inputs to between 0 and 1 because we are inputting the values into a sigmoid function which has a linear range close to 0 and almost flat tails outwards. We would want our values to fall on the linear range rather than being stuck at the tails where the difference would be small and the gradient is virtually zero. In that case, training could be slow or may never catch on. (Also, normalizing the big numbers would simple just making processing throughout the network much easier.)

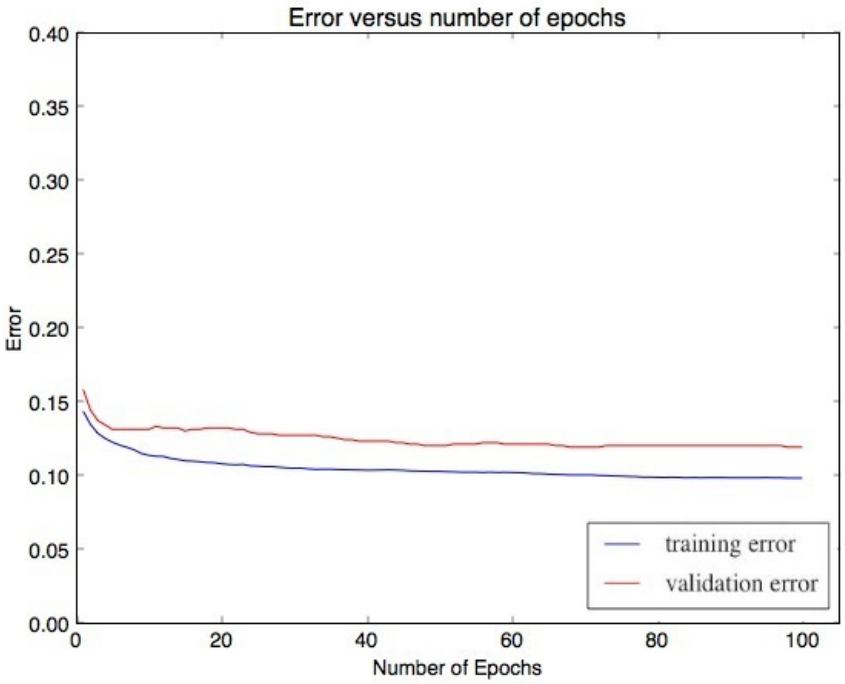
   Problem 6:

   (a) we choose learning rate = 0.1, which performed the best out of the tested learning rates (see table).

   Simple Network with different learning rates:

Learning Rate	Training Performance	Validation Performance
0.001	0.86044444	0.845
0.01	0.88944444	0.864
0.05	0.89900000	0.876
0.08	0.90100000	0.878
0.1	0.90088889	0.880
0.2	0.90133333	0.877
0.5	0.89933333	0.878
1	0.89755556	0.877

   (b) Error graph for a simple neural network:



i. As seen from the graph, the training error and validation error do stop decreasing at some point but don't ever get worse over the course of 100 epochs. Therefore, overfitting appears to not be a big concern. If overfitting was a concern, we would see the training error to continue to decrease as the validation error increase (indicating that the neural network is being modified to be too specific for the training, decreasing the training error, and is not general enough for data in the rest of the set, thus increasing the validation error).

ii. A good number of epochs would be enough epochs for the training error to level off. In this case, it would be about 82 epochs.

iii. Through training, we are trying to build a neural network that can perform well on unseen data. Therefore, it is important not to use test data during any part of the training process so that the test data can truly gauge how well the network would perform on unseen data. Having a validation set allows us to control for overfitting as we tune the neural network while still having a test set of unseen data that can be later used to measure performance on unseen data.

(c) For a step size of 0.1 and for 82 epochs, the training performance is

0.90066667, the validation performance is 0.879, and the test performance is 0.915

Problem 7:

(a) For both 15 hidden units and 30 hidden units, a learning rate of 0.1 performed the best (see tables).

Hidden Network with 15 Hidden Units:

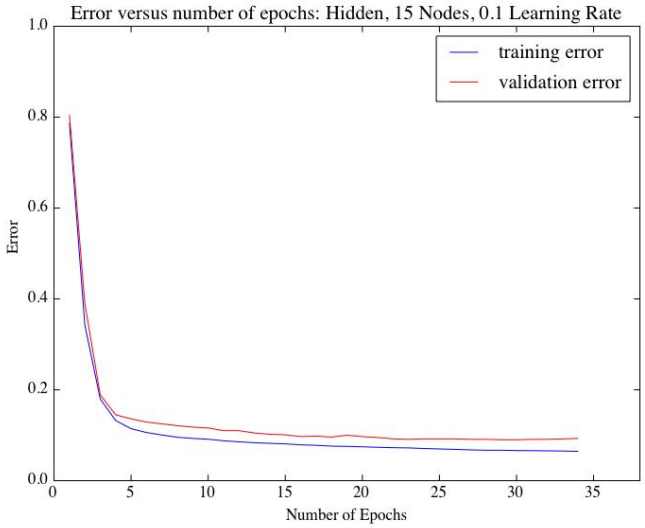
Learning Rate	Training Performance	Validation Performance
0.001	0.21144444	0.193
0.01	0.91311111	0.887
0.1	0.94955556	0.915
1	0.85644444	0.811

Hidden Network with 30 Hidden Units:

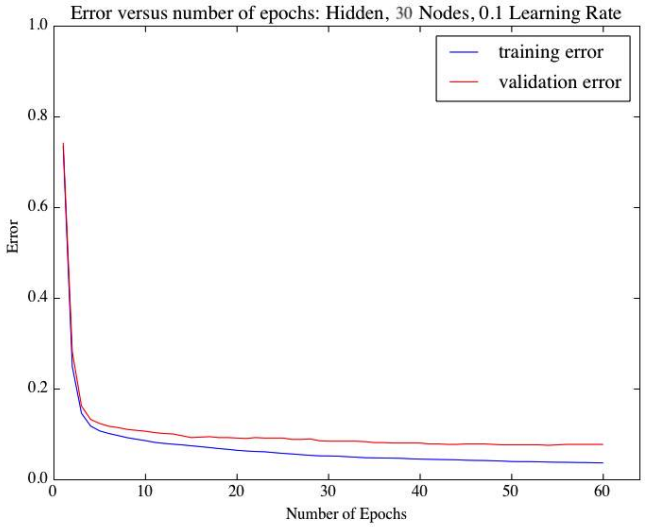
Learning Rate	Training Performance	Validation Performance
0.001	0.65900000	0.684
0.01	0.91722222	0.900
0.1	0.97077778	0.936
1	0.85922222	0.817

(b) We want to implement some sort of stopping protocol that will stop the algorithm when it stops generally improving but that also ignores small ups and downs in the performance. Thus, we will stop the training if the performance on validation data on a certain epoch is not better than the validation performance 15 epochs ago. However, the algorithm will also stop training after 100 epochs so that it does not run for too long in cases where the step size is very small.

(c) Graphs for training set and validation set for the hidden networks:



Testing Performance: 0.932



Testing Performance: 0.942

(d) We used 34 epochs for the hidden network with 15 hidden nodes and 60 epochs for the hidden network with 30 hidden units.

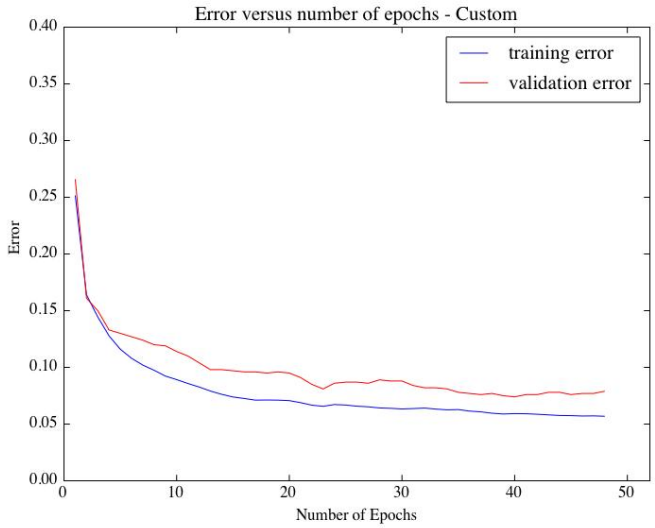
(e) We would choose the hidden neural network structure with 30 hidden units. Even though the structure with 30 units took longer to train than the structure with 15 units, the structure with 30 units had a much better test performance and validation performance. (0.942 vs 0.932 and 0.936 vs 0.915 at a learning rate of 0.1 respectively).

(f) The test set performance for the hidden network with 30 hidden units with cut off of 15 steps is 0.942.

Problem 8:

(a) Our custom network structure is based on the observation that if a digit is divided into 4 quadrants, many of the features of digits reside in one of the four quadrants (for example, the curves of the 3, the "hat" of the 5, the tail of the 2). Thus instead of having a fully connected hidden layer, we made 4 sets of 10 nodes with each of the sets connected to the inputs corresponding to particular quadrant of the image. The hidden nodes are then fully connected to the output nodes. We wanted to test if feature extraction sequestered on particular parts of the image could improve performance while decreasing complexity by having fewer connections between nodes.

(b) After running our custom network for 48 epochs with 10 hidden nodes per quadrant (40 nodes total), the training performace was 0.94311111, the validation performance was 0.921, and the test performance was 0.955.



For a fully connected hidden layer with 15 nodes, there are $196 * 15 = 2940$ connections between the input and hidden layers and $15 * 10 = 150$ connections between the hidden and output layers for a total of 3090 connections. For a fully connected hidden layer with 30 nodes, this becomes 6180 total connections. If we have a partially connected hidden layer with 10 nodes per quadrant, there are $4 * 49 * 10 = 1960$ connections between the input and hidden layers and 400 connections between the hidden and output layers for a total of only 2360 connections.

This means there are much fewer calculations during feed forward and back propagation, which is why this algorithm ran faster than the other two with fully connected hidden layers. The test performance for this structure was also slightly better than that for both the 15-node complete hidden layer network and the 30-node complete hidden layer network. This is likely because it could extract more features with more hidden nodes and more relevant features by focusing on each quadrant independently. Thus, the custom structure is both faster and performs better than the complete hidden layer structures.

4. (a) We might want to use this error function to penalize for heavier weights. The network could be trained to the point where it overfits with a complex network of big weights, or simply keeps training by increasing overall weight. This error function attempts to keep the overall sum of the weights in the network low by factoring the overall magnitude of weights into the error function.

   (b) Apply gradient descent to the new error function:

$$\frac{\partial}{\partial \omega_{current}} C(\vec{\omega}, \vec{x}, y) = \frac{\partial}{\partial w_{ji}} [(y - h_{\vec{w}}(\vec{x}))^2 + \lambda \sum_{m=1}^{M} \sum_{k=1}^{K} \vec{\omega}_{km}^2 + \lambda \sum_{m=1}^{M} \sum_{j=1}^{J} \vec{\omega}_{mj}^2]$$

   The first part of this is identical to the previous loss function so that partial derivative is the same as before. For the sums, only the term with the particular ω is non-zero. Thus the partial becomes:

$$\frac{\partial}{\partial \omega_{mj}} C(\vec{\omega}, \vec{x}, y) = -2(y - h_{\vec{\omega}}(\vec{x}))\sigma'(\vec{x}^T \vec{\omega}) x_j + 2\lambda \omega_{mj}$$

   The -2 becomes absorbed into the learning rate, so the formula for updating the weights becomes:

$$\omega_{mj}^{(r+1)} \leftarrow \omega_{mj}^{(r)} + \alpha[(y - h_{\vec{\omega}}(\vec{x}))\sigma'(\vec{x}^T \vec{\omega}) x_j - \lambda \omega_{mj}]$$