CS181 Assignment 2

Joy Ming and Alisa Nguyen (25 February 2013)

1. Perceptron activation rule

   (a) Bright-or-dark is not able to be recognized by a perceptron. One possibility is to have the weights of all $n = 9$ pixels be initialized to $1/n = 1/9$. Then, multiplying the value if the pixel is on (1) or off (0) with the weights will give something greater than 0.75 if at least 75% pixels are on and will give something less than 0.25 if at least 75% pixels are off. However, this function of returning positive when the value is greater than 0.75 and less than 0.25 is piecewise and not differentiable, therefore not able to be determined by a perceptron. There cannot exist a perceptron that recognizes this feature because this feature is not linearly separable.

   (b) Top-bright is able to be recognized by a perceptron. This perceptron would have $1/3$ weights for the top three pixels and $-1/6$ weights for the bottom pixels. If both have the same fraction of pixels on in both rows then the result should be 0. However, if the top has a larger fraction it would be positive and if the bottom has a larger fraction it would be negative. Therefore, simply checking to see if the result is greater than zero can tell if a larger fraction of pixels is on in the top row than in the bottom two rows.

   (c) It is not possible to present a perceptron that can check to see if a set of pixels that are on is connected because it is not possible to model when different points of data are correlated. Suppose for contradiction that there is a perceptron that could check to see if a set of pixels that are on is connected. Then that given perceptron would be able to determine data that is not linearly separable, which is not possible. Therefore, there does not exist a perceptron that is able to see if a set of pixels is connected.

2. Four different possible learning algorithms

   (a) Decision trees are most effective when there are a few specific characteristics to split on with different values. However, it is difficult in this case to even determine which characteristics to split on, even if there were some way to implement continuous attributes. One possibility for splitting would be starting from a given pixel such as the top left pixel to see if its neighbors are activated or not. However, this would result in a lot of branches for each pixel and its neighbors. This could also be difficult to train and determine these splits. Therefore the fact that few attributes determine decisions and that they do not construct new features makes decision trees less robust in digit recognition. However, decision trees are useful in the cass of fast learning and easy to interpret models.

   (b) Boosted decision stumps are especially useful for making prediction based on the value of a single input feature over a specific threshold. Though these decision stumps can reach a higher level of accuracy after boosting, their level of simplicity that contrasts that of the complicated features of the decision trees is still not well adapted for the problem. One way to approach this would be to have a separate decision stump for each pixel. However, this would result in it being very difficult to figure out the linking of the different, separate pixels. Combatting this inability to consider connected features would be to track the presence of certain patterns. However, it would still be difficult to even put the frequency of these patterns together as well.

   (c) Perceptrons are better than decision trees and decision stumps at learning about the set of data as opposed to individual pixels. This is because the perceptrons are looking at the sum of the different pixels. However, perceptrons do have some weaknesses, as evidenced by the previous problems. They are still limited only to linearly separable data and their decision boundaries

have to be based on differentiable functions that will not be able to consider piecewise functions or connectedness. Therefore, though perceptrons are a step toward understanding, there have to be more layers to get a better grasp of the data.

(d) Multi-layer feed-forward neural networks is what we implemented in the problem set, which speaks to its validity as a means of approaching the problem of digit recognition. This problem has as many as 196 pixels for a 14x14 image, as spoken about many times in discounting the other methods. The multi-layer feed-forward network can help with this in that it can take many different inputs. The problem of digit recognition also has 10 distinct output categories, which is well embodied by the various output nodes in the multi-layer feed forward neural networks. One drawback is that this needs a lot of previous knowledge and learns fairly slowly.

3. Implementing a neural network

5. We normalize the input values from [0, 255] to between 0 and 1 so we are able to better utilize the logistic sigmoid function.

6. Experiment with simple networks

(a) We used a learning rate of 0.1 based on a running 15 epochs of four different learning rates:

- Validation and training performance for a learning rate of 1.0 for different epochs:

```
* * * * * * * * *
Parameters => Epochs: 15, Learning Rate: 1.000000
Type of network used: SimpleNetwork
Input Nodes: 196, Hidden Nodes: 0, Output Nodes: 10
* * * * * * * * *
1 Performance: 0.34900000 0.340
2 Performance: 0.47555556 0.449
3 Performance: 0.55900000 0.529
4 Performance: 0.56588889 0.540
5 Performance: 0.69877778 0.675
6 Performance: 0.71755556 0.693
7 Performance: 0.79322222 0.782
8 Performance: 0.74711111 0.728
9 Performance: 0.75333333 0.731
10 Performance: 0.71777778 0.704
11 Performance: 0.71500000 0.709
12 Performance: 0.72066667 0.707
13 Performance: 0.76622222 0.762
14 Performance: 0.70933333 0.703
15 Performance: 0.78533333 0.768
```

- Validation and training performance for a learning rate of 0.1 for different epochs:

```
* * * * * * * * *
Parameters => Epochs: 15, Learning Rate: 0.100000
Type of network used: SimpleNetwork
Input Nodes: 196, Hidden Nodes: 0, Output Nodes: 10
* * * * * * * * *
1 Performance: 0.55100000 0.514
2 Performance: 0.65055556 0.608
3 Performance: 0.71222222 0.690
```

```
4 Performance: 0.75144444 0.744
5 Performance: 0.77644444 0.764
6 Performance: 0.79433333 0.787
7 Performance: 0.80655556 0.797
8 Performance: 0.81500000 0.810
9 Performance: 0.82177778 0.815
10 Performance: 0.82666667 0.819
11 Performance: 0.82966667 0.825
12 Performance: 0.83422222 0.827
13 Performance: 0.83622222 0.828
14 Performance: 0.83733333 0.829
15 Performance: 0.84055556 0.830
```

- Validation and training performance for a learning rate of 0.01 for different epochs:

```
* * * * * * * * *
Parameters => Epochs: 15, Learning Rate: 0.010000
Type of network used: SimpleNetwork
Input Nodes: 196, Hidden Nodes: 0, Output Nodes: 10
* * * * * * * * *
1 Performance: 0.27844444 0.296
2 Performance: 0.36988889 0.370
3 Performance: 0.40066667 0.388
4 Performance: 0.45244444 0.441
5 Performance: 0.49977778 0.473
6 Performance: 0.54100000 0.516
7 Performance: 0.57144444 0.548
8 Performance: 0.60288889 0.575
9 Performance: 0.62600000 0.596
10 Performance: 0.64577778 0.614
11 Performance: 0.66522222 0.630
12 Performance: 0.68511111 0.644
13 Performance: 0.70066667 0.670
14 Performance: 0.71355556 0.686
15 Performance: 0.72444444 0.702
```

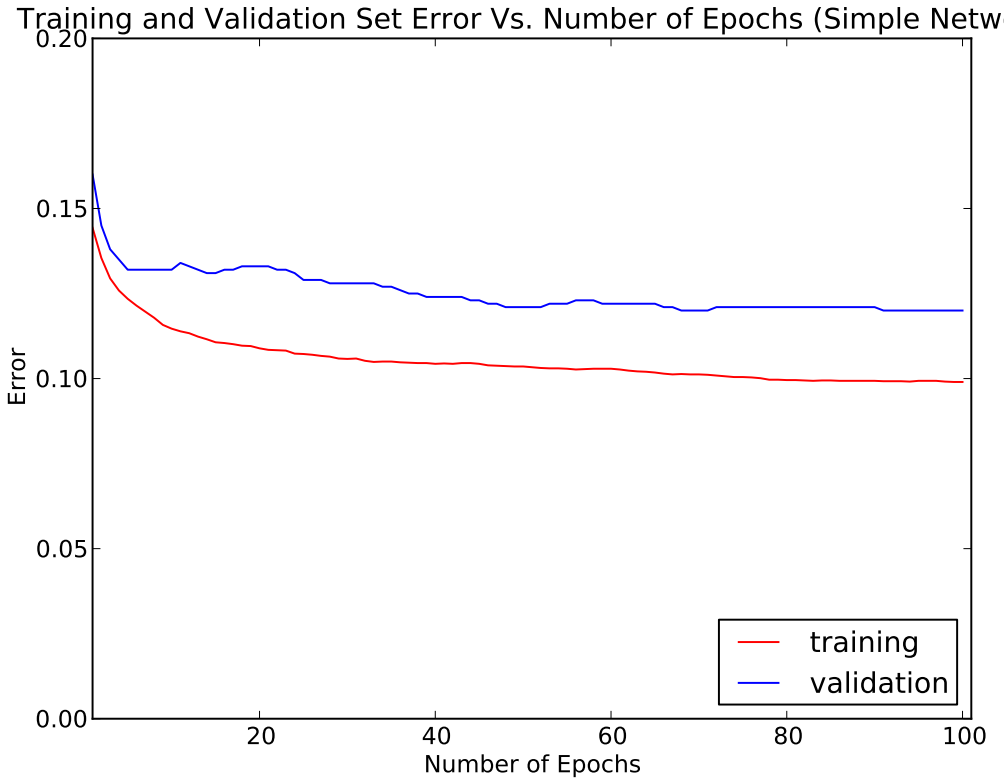
- Validation and training performance for a learning rate of 0.001 for different epochs:

```
* * * * * * * * *
Parameters => Epochs: 15, Learning Rate: 0.001000
Type of network used: SimpleNetwork
Input Nodes: 196, Hidden Nodes: 0, Output Nodes: 10
* * * * * * * * *
1 Performance: 0.11444444 0.090
2 Performance: 0.12455556 0.100
3 Performance: 0.17788889 0.162
4 Performance: 0.20377778 0.191
5 Performance: 0.22088889 0.208
6 Performance: 0.23300000 0.215
7 Performance: 0.24100000 0.226
8 Performance: 0.24988889 0.231
9 Performance: 0.25900000 0.241
10 Performance: 0.26888889 0.252
```

```
11 Performance: 0.27933333 0.264
12 Performance: 0.28944444 0.269
13 Performance: 0.29755556 0.274
14 Performance: 0.30788889 0.281
15 Performance: 0.31344444 0.289
```

We chose to use 0.1 as the learning rate because within the 15 epochs, it scored consistently the highest and seemed to be moving most steadily toward progress.
NOTE: These tests were actually run on the program before the bug mentioned in Piazza Post 105 of initializing the transformed values of each input node to be equal to the raw value was fixed. However, since these results are relative to each other, we kept the data just to show which learning rate had the best performance.

(b)



i. Yes, we are in danger of overfitting by training for too many epochs as mentioned in lecture. The ways we discussed how to combat this include early stopping, model selection, weight pruning, regularization, and weight sharing.

ii. A good number of epochs to train for is around 25, as we determined experimentally. It is important that we use a validation set rather than the actual test set to tune the number of epochs because FILL IN

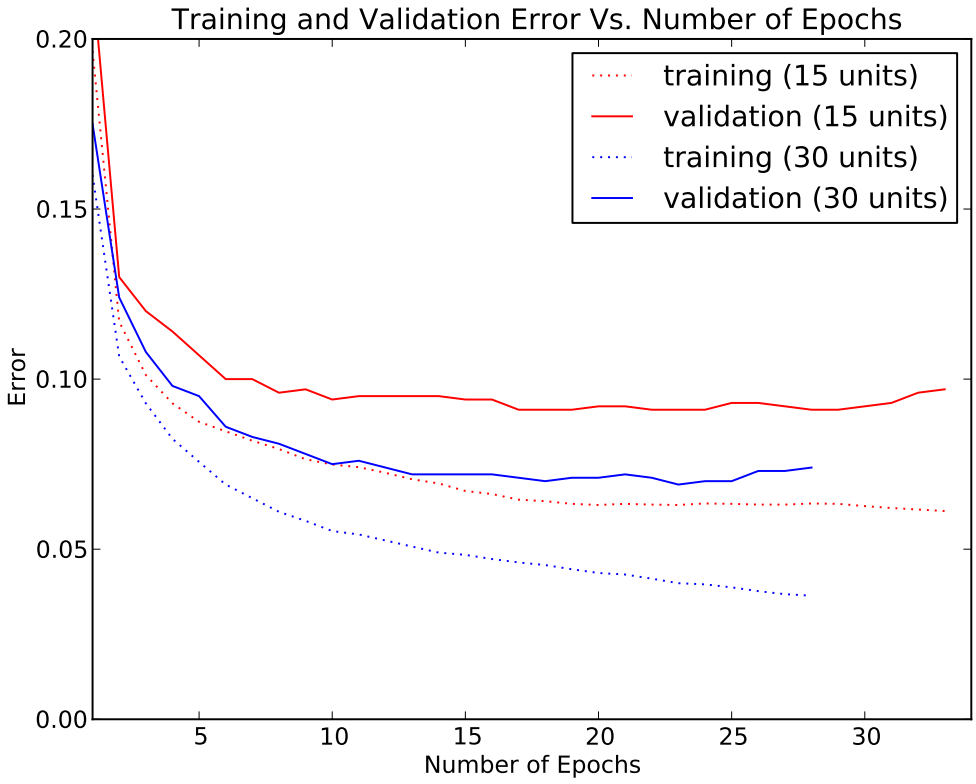
(c) The training performance is 0.901.
The validation performance is 0.880.
The test performance is 0.913.

7. Experiment with networks with a single hidden layer

(a) We decided that we wanted to use a learning rate of 0.3 for both 15 and 30 fully connected units in the hidden layer based on previous experimental data.

(b) We determine when to stop training by checking the difference between the validation performance and the previous validation performance. If the difference is less than a given ϵ, which in this case was 0.001, at least five times in a row, then stop the training.

(c)



(d) We used 33 epochs for 15 hidden units and 28 epochs for 30 hidden units.

(e) Based on these experiements, we would use the network structure with 30 fully connected hidden units because both training set error and validation set error were lower for the 30 fully connected hidden units compared to the 15. By the 28th epoch round for the 30 hidden units, the training set error was 0.036 and the validation set error was 0.074, which is considerably less than the errors for 15 units by the 33rd round, which were 0.612 and 0.097 respectively.

(f) The test set performance of the network with 30 hidden units for 28 epochs was $\boxed{0.949}$. This is higher than the test set performance of the network with 15 hidden units for 33 epochs, which was 0.939. This compares to the committee of perceptrons by FILLIN.

8. CustomNetwork

(a) We chose to implement more hidden layers to see how this impacts the performance of the neural networks. First we put in two hidden layers of 15 then we put in three hidden layers of 10.

(b) Our new network's test performances include:

- For a neural network with 2 hidden layers of 15 node each using learning rate of 0.3 and the convergence stopping algorithm:

```
* * * * * * * * * *
Parameters => Epochs: 10, Learning Rate: 0.300000
Type of network used: CustomNetwork
Input Nodes: 196, Hidden Nodes: 30, Output Nodes: 10
* * * * * * * * * *
1 Performance: 0.10177778 0.114 0.104
2 Performance: 0.10077778 0.098 0.106
3 Performance: 0.10077778 0.098 0.106
4 Performance: 0.10077778 0.098 0.106
5 Performance: 0.10077778 0.098 0.106
```

- For a neural network with 3 hidden layers of 10 nodes each using learning rate of 0.3 and the convergence stopping algorithm:

(c) It was surprising, at first that the performances were so drastically low to begin with and they stopped pretty quickly after the minimum five rounds, meaning they did not improve significantly. However, thinking more about the data, it seemed to make more sense

4. Changing error function

(a) Error function C is implementing the idea of regularization, or that the less complex solution, or one that has less dimensions or degrees, is more desirable others that might work. In the new error function, different values are added onto the original loss function that will increase the sum depending on the increased complexity or layers of the network. This will penalize the loss function by squared weight values, making it more crucial to create less complicated graphs to minimize the loss function. This looks like the regularization example:

$$\text{Cost}(w) = L(w; D) + \lambda \cdot \text{Complexity}(w)$$

(b) Our weight update rule for the error function C derived from gradient descent is:

$$w_k^{(r+1)} \leftarrow w_k^{(r)} + \alpha(-1\lambda w_k + x_k\delta)$$