CS 181 Assignment 2: Neural Networks

Mark VanMiddlesworth & Shuang Wu

2/25/11

PROBLEM 1. We let $\mathbf{x} \in \{1, -1\}^9$ be the input vector of pixel states, and $\mathbf{w} = \{w_0, w_1, \ldots, w_9\}$ the weights of the 3×3 array of pixels arranged as follows:

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

with w_0 the threshold.

1. **Bright-or-dark:** Suppose there exists a perceptron with weights $\mathbf{w}$ that recognizes this feature. When pixels x_1 and x_2 are on and the remainder off, more than 75% of the pixels are off, so the output is 1 and we have

$$w_0 + w_1 + w_2 - w_3 - \cdots - w_9 > 0. \tag{1}$$

   When pixels x_1, x_2, and x_3 are on and the remainder off, we do not have 75% of the pixels off or on, so the output is -1 and we have

$$w_0 + w_1 + w_2 + w_3 - w_4 \cdots - w_9 \leq 0. \tag{2}$$

   Combining (5) and (6), we get $w_3 < 0$. Similarly, when pixels x_1 and x_2 are off and the remainder on, we have

$$w_0 - w_1 - w_2 + w_3 + \cdots + w_9 > 0, \tag{3}$$

   and when x_1, x_2, and x_3 are off and the remainder on, we have

$$w_0 - w_1 - w_2 - w_3 + w_4 + \cdots + w_9 \leq 0. \tag{4}$$

   Combining (7) and (8), we get $w_3 > 0$, ※. Hence no such perceptron exists.
2. **Top-bright:** We have the perceptron with weights $w_1 = w_2 = w_3 = 2$ and $w_4 = \ldots = w_9 = -1$. If we set $w_0 = 0$, then $\mathbf{w} \cdot \mathbf{x} > 0$ if and only if $2(x_1 + x_2 + x_3) > x_4 + \cdots + x_9$, which occurs only when a greater fraction of pixels are on in the top row than in the bottom two rows.
3. **Bonnected:** Suppose there exists a perceptron with weights $\mathbf{w}$ that recognizes this feature. Suppose we have pixels x_2, x_4, and x_6 off, and everything else on. This is disconnected, so

$$w_0 + w_1 - w_2 + w_3 - w_4 + w_5 - w_6 + w_7 + w_8 + w_9 \leq 0. \tag{5}$$

   Now suppose we have pixels x_1, x_3, and x_5 off, and everything else on. This is also disconnected, so

$$w_0 - w_1 + w_2 - w_3 + w_4 - w_5 + w_6 + w_7 + w_8 + w_9 \leq 0. \tag{6}$$

   The image with x_7, x_8, x_9 on and everything else off is connected, so

$$w_0 - w_1 - w_2 - w_3 - w_4 - w_5 - w_6 + w_7 + w_8 + w_9 > 0 \tag{7}$$

and the image with everything on is also connected, so

$$w_0 + w_1 + w_2 + w_3 + w_4 + w_5 + w_6 + w_7 + w_8 + w_9 > 0. \tag{8}$$

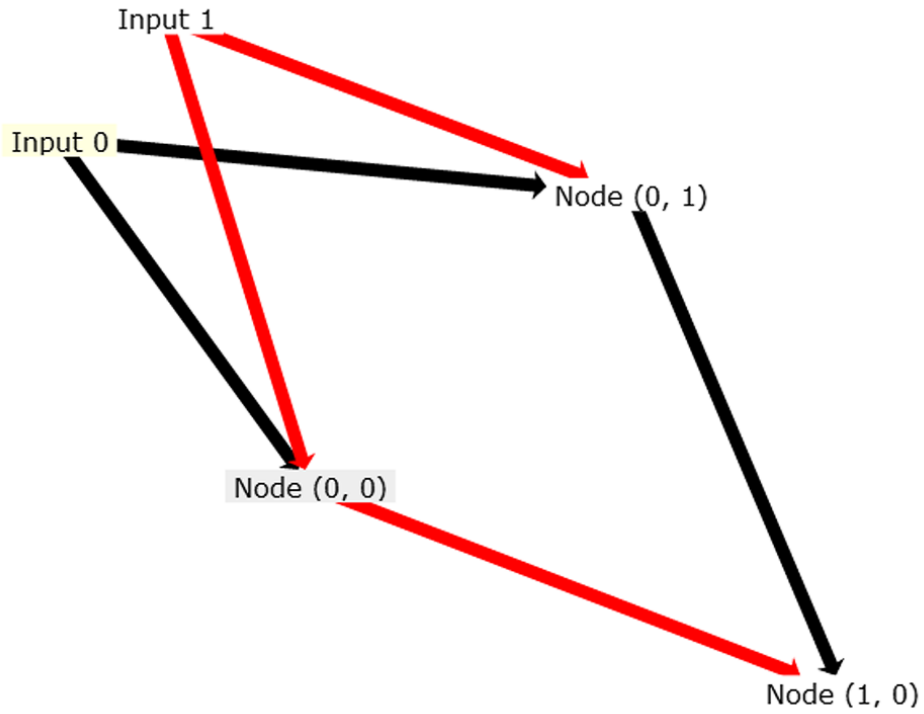
Combining (5) and (7), we have $w_1 + w_3 + w_5 < 0$, and combining (6) and (8), we have $w_1 + w_3 + w_5 > 0$, ※. Hence no such perceptron exists.

Problem 2.

1. **Decision trees:** Decision trees would be ill-suited to digit recognition. In this scenario, single attribute values are far less important than the relationship between attributes (adjacent pixels), a task that decision trees are particularly weak at. Since it is unlikely that a few pixels determine the label of an image (especially if the input digits are of slightly different sizes and orientations), decision trees will split on a lot of pixels, resulting in huge trees and likely overfitting the training data. Decision trees could perhaps be made to work if a) there is very little variation in each appearance of a digit (in size, shape, and orientation) or b) if higher-level features (things like edge counts, number of loops, etc.) were extracted and used for training in place of pixel levels. The former is unlikely (and would make the task easy to begin with if it were the case), and the latter would likely require significant computational complexity.
2. **Boosted decision stumps:** Without any preprocessing, this would also be a poor approach since decision stumps would likely be too weak as base learners, as they only split on a single pixel. This is made even weaker when the digit images are of different orientations and sizes, so that single pixels are very unlikely to have any significance in classifying digits. However, with the proper preprocessing of data to extract higher level features, boosted decision stumps could perform quite well, as seen in lecture with the Adaboost+C4.5 algorithm on 16 preprocessed attributes. Again however, this would require some computational complexity to extract these features.
3. **Perceptrons:** This approach would be better than decision trees and boosted decision stumps, and would probably be moderately successful if we used the proper output encoding. In a distributed encoding, with one perceptron per digit, each perceptron would attempt to learn to recognize a single digit based on all pixels in the input image. If the images for each digit have relatively similar sizes, shapes, and orientations, we would expect the images to have roughly the same areas of light and dark pixels, which perceptrons would be able to recognize. With images that are skewed or rotated however, perceptrons would learn poorly without pre-processing to normalize the inputs.
4. **Multi-layer feed-forward neural networks:** This is the best approach of the four without preprocessing of the data. Neural networks consider all input pixels, which is suitable for this task since most pixels (and more importantly, the relationship between most pixels) are significant in determining the class of the input. Neural networks can detect non-linearly separable relationships between pixels when hidden node layers are used, so they are able to detect transformations such as rotations or translations. Although neural networks are slow to learn, once trained they can be efficiently stored and used to classify images based on raw pixel input without any preprocessing.

Problem 3.

2. **XOR:** The network we generated is given below:



and has the following weights:

```
Perceptron([2.71345303682141, -2.760176633812281], 2.303962802509964, 0)
Perceptron([3.3972873412750366, -3.5996622402680525], -3.9311904233798027, 1)

Perceptron([-5.175225209852754, 5.19590443203013], 2.405813513795126, 0)
```

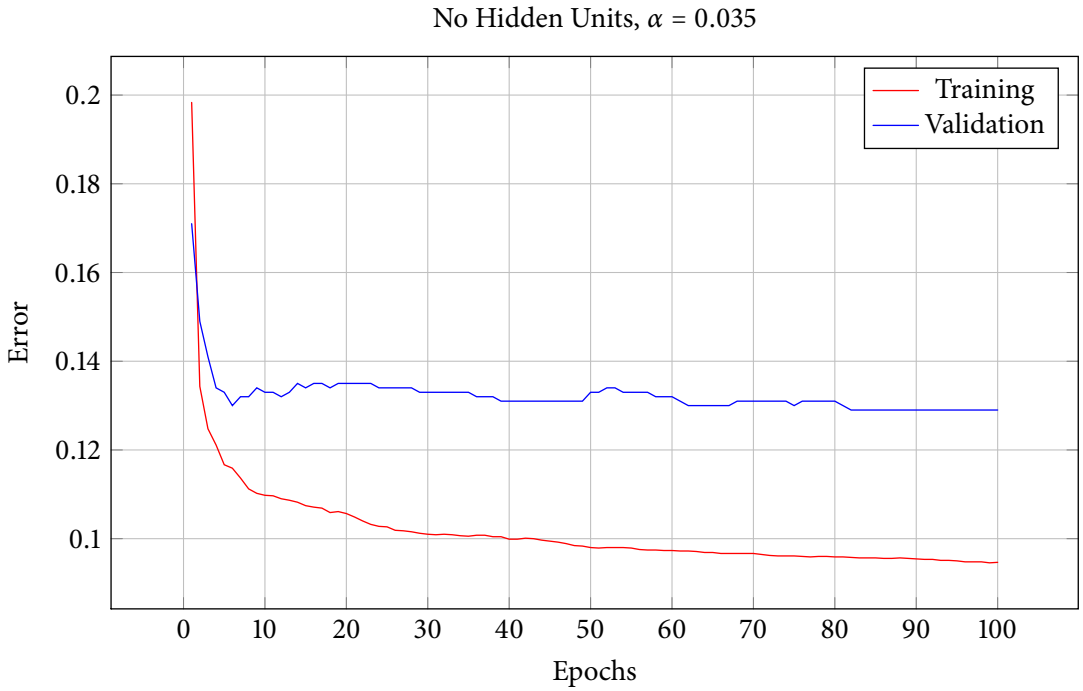
Note that node (0,0) has a positive w_0, positive w_1 and negative w_2; node (0,1) has a negative w_0, positive w_1 and negative w_2; and node (1,1) has a positive w_0, negative w_1 and positive w_2. Intuitively, we may consider a negative weight on a boolean input to be a logical negation. Since our perceptrons all accept two inputs, we may consider w_0 to indicate either a logical $\wedge$ or $\vee$ operation; that is, does it classify values of *in* near 0 to be positive ($\vee$) or negative ($\wedge$)? Indeed, if we run through the truth tables for each of these perceptrons, we see that node (0,0) is equivalent to $a \vee \neg b$, node (0,1) is equivalent to $a \wedge \neg b$, and node (1,1) is equivalent to $\neg a \vee b$. Hence, this network is equivalent to

$$\neg(a \vee \neg b) \vee (a \wedge \neg b) = (a \wedge \neg b) \vee (b \wedge \neg a) = a \oplus b.$$

When re-running *Solve XOR Part II* many times, it occasionally fails to classify one of the inputs incorrectly. This could be because the network failed to converge before the training stopped. In general, tuning the number of training rounds and the learning rate can alleviate this problem: the higher the number of epochs, the more likely a convergence will be reached, and learning rates which are too high are prone to oscillation around a minimum, while learning rates that are too low will take longer to converge. Another reason why the network might fail to converge is if it gets stuck in a local minimum; this may be alleviated

by initializing with random weights for each run and taking the best performer on a validation set out of several runs.

3. **Experimental Analysis:**
   (a) We used a learning rate of 0.035. We selected this rate by training a network over 100 rounds on the first 2000 entries of the `training-9k.txt` file (instead of the default `training-2000.txt` file, so that we are using the same number of inputs when we switch over to the full training set). We trained with learning rates in the range $(0, 2]$ in increments of 0.1, selected the learning rate that had the best maximum single round validation performance within those 100 rounds, then trained with learning rates around that value in increments of 0.01, etc. Completing this process several times, we converged to an optimal training rate of 0.035 on 3 out of 4 runs. We then trained a network with that learning rate on the full training set for the remaining parts of this question.
   (b) We have the following results from training:



   i. We do not appear to be in danger of significant overfitting by training for too many epochs, at least in the range that we are looking at. The validation error does not increase with more epochs as would be the case if overfitting occurs. However, it appears that the training error is still decreasing as we near 100 epochs, so if we continued training for more rounds, we might eventually see some overfitting. However, the validation error appears to be relatively stable, so any overfitting is unlikely to be significant.
   ii. There is no improvement in validation error beyond 82 rounds, so it would be a good number of epochs to train for.
   iii. It is important that we use a validation set to tune the number of epochs rather than the test set in order to avoid unintentional overfitting. The number of epochs we train for is a parameter of our training; if we were to tune that parameter to our test set, then we are essentially optimizing

our training parameters based on the test set. We must keep the test set separate from any data used to train a learning algorithm if we want the test performance to be indicative of expected performance.

(c) At 82 epochs and a learning rate of 0.035, the training performance is 0.904, the validation performance is 0.871, and the test performance is 0.928.

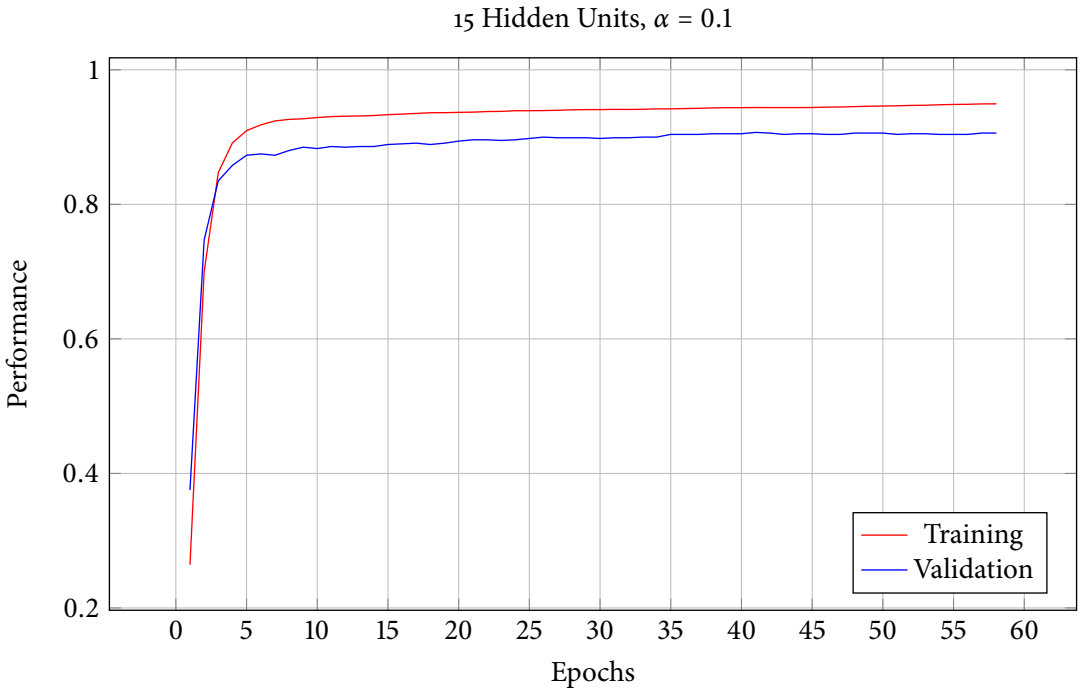
4. **Model Selection:**

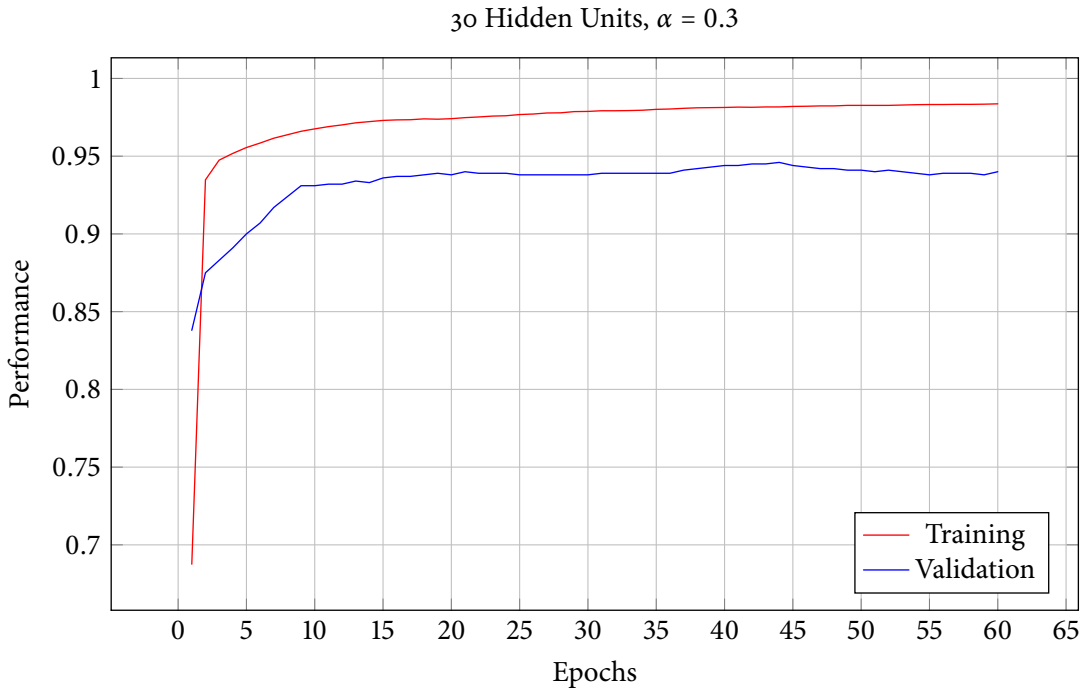
(a) Distributed encoding performs better than binary encoding. This is in line with what we know about encodings, since distributed encoding fully separates each target value. Binary encoding on the other hand, will share positive outputs for different target values. When classifying digits, there is no reason to believe that three (binary output `0011`) and one (`0011`) are any more related to each other than eight (`0011`).

(b) We trained the 15 hidden layer network at a learning rate of 0.1, and the 30 hidden layer network at a learning rate of 0.3.

(d) We keep track of the best validation performance thus far, and compare the current validation to that value. If validation performance has dropped by more than a threshold value from the maximum or it has been more than a threshold number of rounds since we saw the maximum, we stop training. The threshold is calculated as ten times the maximum number of training rounds (the multiple of ten was chosen during testing to yield reasonable training lengths) divided by the current round, so that it is harder to stop earlier on in the training when we are more susceptible to large jumps in performance.

(e) The performance graphs are below:





(f) Our algorithm stopped our training at 58 epochs for 15 hidden units, and 60 epochs for 30 hidden units

(g) The test performance for 15 hidden units is 0.940, the test performance for 30 hidden units is 0.965. This does agree more or less with validation set performance (in fact, it is better on both counts, probably because the data was chosen to be the "cleaner testing data" as explained in the file descriptions). Based on these experiments, we would choose 30 hidden units since it performs significantly better than both 15 hidden units and no hidden units, without sacrificing too much running time.

PROBLEM 4.

1. The alternative error function contains two parts: the original gradient descent error function $E(\mathbf{w})$, and the extra term

$$\gamma \sum_{j \in V} \sum_{i \in \text{Parents}(j)} w_{ji}^2.$$

This term is just the sum of squared weights multiplied by a factor γ. $F(\mathbf{w})$ is trying to address the problem of overfitting by placing a penalty on the sum of squared weights, since single heavily-weighted inputs often indicate overfitting when compared to multiple moderately-weighted inputs. Since we are squaring weights, large weights have an added penalty.

2. We compute

$$\begin{aligned}\frac{\partial F}{\partial w_{ji}} &= \frac{\partial}{\partial w_{ji}}\left(E + \gamma \sum_{j\in V} \sum_{i\in \text{Parents}(j)} w_{ji}^2\right) \\ &= -\sum_{d=1}^{n} a_{id}\delta_{jd} + \gamma \sum_{j\in V} \sum_{i\in \text{Parents}(j)} \frac{\partial}{\partial w_{ji}}(w_{ji}^2) \\ &= -\sum_{d=1}^{n} a_{id}\delta_{jd} + 2\gamma \sum_{j\in V} \sum_{i\in \text{Parents}(j)} w_{ji}.\end{aligned}$$

We absorb the factor of 2 into γ, and our update rule is then

$$w_{ji}^{(t+1)} \leftarrow w_{ji}^{(t)} + \alpha\left(\sum_{d=1}^{n} a_{id}\delta_{jd} - \gamma \sum_{j\in V} \sum_{i\in \text{Parents}(j)} w_{ji}^{(l)}\right)$$