CS 181: Assignment 1

Angeline Baniqued & Michael Wee

1. Perceptrons

(1) bright-or-dark (At least 75% of the pixels are on, or at least 75% of the pixels are off.)

A perceptron cannot recognize this feature. A perceptron can easily be trained to recognize one of the two conditions, for example by having all of the weights be 1 and having a threshold activation function of either 7.5 or 2.5. However, once it is set to recognize one, it cannot recognize the other because the threshold would have been set at a hard limit already.

(2) top-bright (A larger fraction of pixels is on in the top row than in the bottom two rows)

A perceptron can recognize this feature. We can see this by having the three weights on the top row be $\frac{1}{3}$ and all the other six weights be $-\frac{1}{6}$. We would set the threshold to 0 and check if the sum is greater than 0 to indicate a positive example and otherwise a negative example.

(3) connected (The set of pixels that are on is connected. (In technical terms, this means that if we define a graph in which the vertices are the pixels that are on, and there is an edge between two pixels if they are adjacent vertically or horizontally, then there is a path between every pair of vertices in the graph.)

A perceptron cannot detect this feature. There are too many variations and combinations in terms of number of pixels that are on as well as the location of where these pixels are. For example, 1 or 2 pixels on could be a path, as can 8 or 9. We cannot do anything with the total number of pixels on. We also cannot learn weights for specific squares or specific areas of the grid because paths can appear in any location. So any way we train the weights we can come up with a counter example path that the weight set incorrectly classifies by countering the training strategy. This means either by changing the number of pixels that are on, or constructing a path on a different portion of the grid than where the weights were originally tuned to correctly classify paths.

2. Learning Algorithms

(1) The domain of handwritten digit recognition is 14x14 pixels with intensities ranging from 0 to 255. The features for this domain are the pixel intensities themselves.

- Decision trees

  Decision trees that are modified to split on ranges of continuous values would be able to classify handwritten digits though they will neither be as effective nor as natural for this domain as other techniques such as using neural networks. The decision tree would need an effective algorithm to take into account lookahead because digit recognition revolves around combinations of pixel intensities. An algorithm analogous to ID3 would not work well because of the reliance of attributes on one another and the fact that nearby pixels affect recognition of what digit is formed. Because of the relatively large number of pixels, an effective decision tree would have to be deep because it would have to predict on combinations of features, not single attributes. Decision trees would also have to be pruned since their tendency to overfit combined with many features (as well as the possibility to split more than once on any given continuous attribute). While decision trees conceivably

could be constructed to do well on digit recognition, it does not lend itself naturally to this domain. An analogous algorithm to ID3 would fail miserably. So there are decision trees that can classify the digits and were sure there may be algorithms that adequately do lookahead and attribute combinations, but there are more natural choices for a domain that requires combinations of features to be effective.

- Boosted decision stumps

  Boosted decision stumps wouldnt do well on this domain. Because boosted decision stumps split on only one attribute, they do not combine features and individually will not be able to capture patterns in the data well enough to classify digits. Individual stumps can do no better than chance since on their own they cannot predict what digit a particular pixel intensity is a part of.

- Perceptrons

  Perceptrons would be especially bad for recognizing digits. Since there are 10 possible digits to correctly classify, it presents too much variability for a single perceptron to classify. Perceptrons have more limitations beyond just being able to output binary values, which is a problem in a 10-class domain.

- Multi-layer feed-forward neural networks

  Multi-layer feed-forward neural networks are very suited for the domain of recognizing hand written digits. Because the digits are represented as a matrix of pixels, a neural network can propagate pixel intensities and calculate a decision boundary based on combinations of all of these features. Moreover, the flexibility in designing feed-forward networks lends itself even further to classifying digits, allowing for custom hidden layers to compute and pass forward useful features like averaging/subsampling and feature maps downstream.

3. Neural Networks

(1) See FeedForward function.

(2) See Backprop function.

(3) See Train function.

(4) See EncodeLabel, GetNetworkLabel, Convert and InitializeWeights functions.

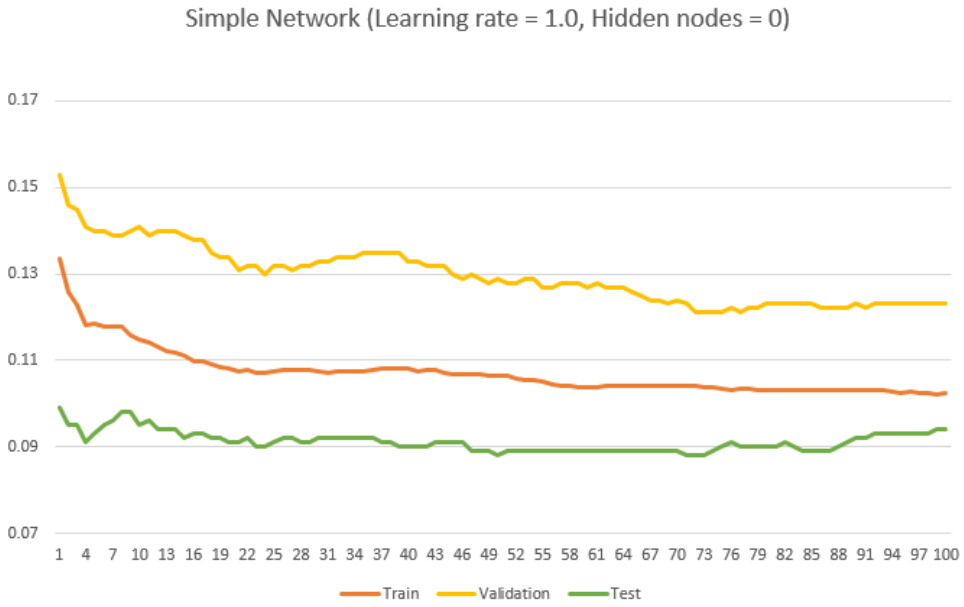
(5) Input data normalization is useful when we think about the shape of the sigmoid function which is the activation function we use. The sigmoid function is approximately linear when input is near 0 and converges to 1 for large input and 0 for small input. Since the activation/sigmoid function takes in the linear combination of weights and activity at each parent node, we want that linear combination to be around the vicinity of 0 which can be achieved by normalizing the input values from [0,255] to between [0,1].

   Furthermore, if we have a wide range of input values, the bigger values will tend to have a higher contribution to the output error, and so, the error reduction algorithm will be focused on the higher values, neglecting the information from the small valued variables.

(6) Simple Network

a) **Chosen learning rate = 1.0**

b) Chart of Training Set and Validation Set Error vs. Number of Epochs



c) Like other statistical models, neural networks are also subject to overfitting. By looking at the plot of the number of epochs of training on the x-axis, and the classification error on the y-axis, we see the standard overfitting phenomenon.

d) From the plot, the classification error on the validation set increases at around epoch 23, so we could train for 23 epochs and then stop there.

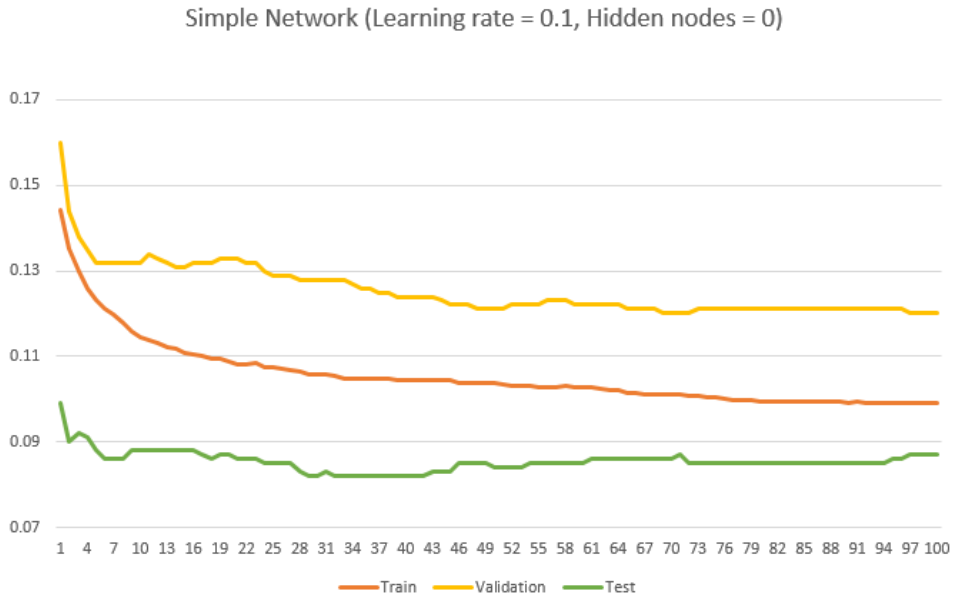
e) It's useful to divide the dataset into three (train, validation and test) because each set serves an important purpose. We need the training set to actually train the neural network, but then during training, the classification error on the training set will continue to decrease as the number of epochs increases. Thus, we need the validation set because there will be some point after which we will be overfitting and we can detect overfitting by looking at when the validation set increases. It's important to use the validation set instead of the test set because (1) we shouldn't be peeking at the test data because the test set should be set aside for when we want to 'test' a non-overfitting neural network and (2) without a validation set, we wouldn't be able to find a non-overfitting neural network to test.

f) This training rate of 1.0 produced the highest performance on the training, validation and test sets:
Training set performance = 0.89
Validation set performance = 0.87
Test set performance = 0.91

For reference and comparison, here are the results of the other learning rates that we didn't end up choosing (0.1, 0.01, 0.001).

a) **Learning rate = 0.1**

b) Chart of Training Set and Validation Set Error vs. Number of Epochs



i) same answer as in the 1.0 learning rate case

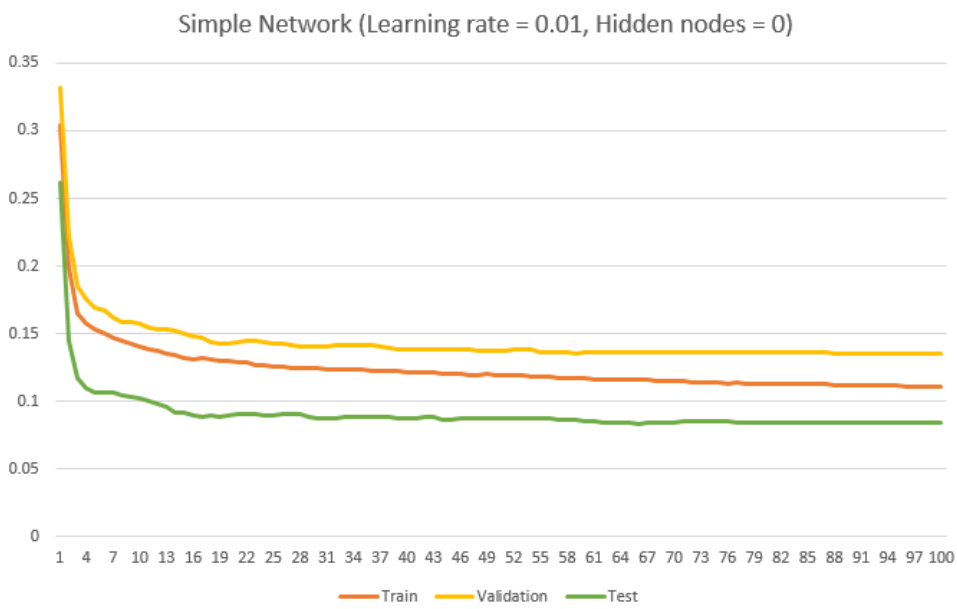
ii) From the plot, the classification error on the validation set increases at around epoch 14, so we could train for 14 epochs and then stop there.

iii) same answer as in the 1.0 learning rate case

c) Training set performance = 0.88
Validation set performance = 0.87
Test set performance = 0.91

a) **Learning rate = 0.01**

b) Chart of Training Set and Validation Set Error vs. Number of Epochs



i) same answer as in the 1.0 learning rate case

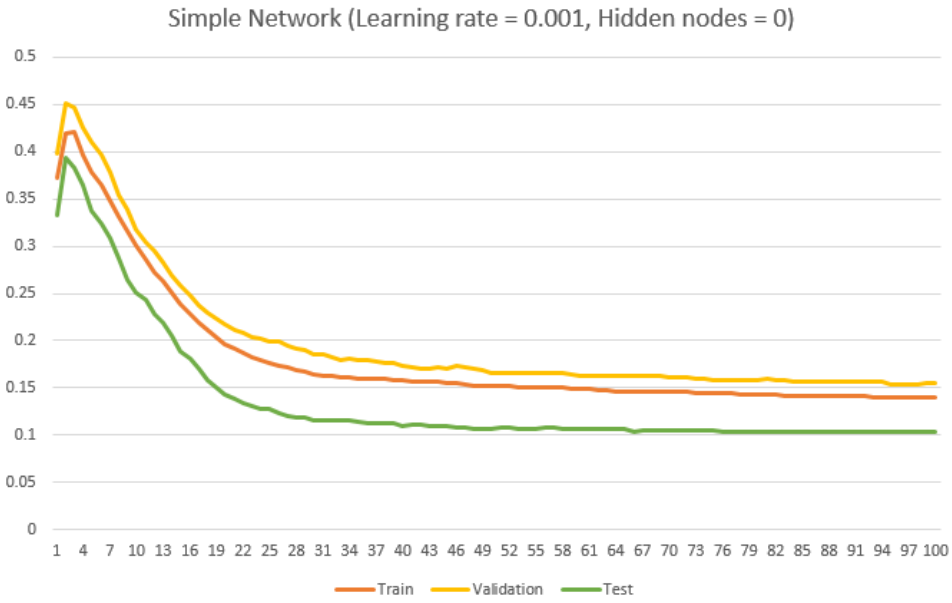
ii) From the plot, the classification error on the validation set increases at around epoch 19, so we could train for 19 epochs and then stop there.

iii) same answer as in the 1.0 learning rate case

c) Training set performance = 0.87
Validation set performance = 0.86
Test set performance = 0.91

a) **Learning rate = 0.001**

b) Chart of Training Set and Validation Set Error vs. Number of Epochs



i) same answer as in the 1.0 learning rate case

ii) From the plot, since the learning rate is so small, we might require a greater number of epochs (greater than 100) to see the overfitting phenomenon more clearly.

iii) same answer as in the 1.0 learning rate case

c) Training set performance = 0.86
Validation set performance = 0.85
Test set performance = 0.90

(7) Hidden Network

a) We used 1.0, 0.1, 0.01, and 0.001 as the learning rates for both 15 and 30 hidden units. We chose these rates in order to find the order of magnitude of the most effective learning rate.

For both 15 and 30 hidden units, we found that the optimal learning rate was on the order of 0.1. Test/train/validation performances are mentioned in the succeeding subproblems. The learning rate of 1.0 caused oscillations in gradient descent and didn't really converge whereas the small learning rates of 0.01 and 0.001 cause gradient descent to converge really slowly.

b) We monitor the validation set performance of the neural network. Our stopping condition is when the validation set performance decreases twice in a row. Empirically we found that a one-time decrease in validation was not enough to guarantee a good stopping point and a three-in-a-row decrease in validation performance rarely occurred in general.

c) Chart of Training Set and Validation Set Error vs. Number of Epochs (15 hidden units)

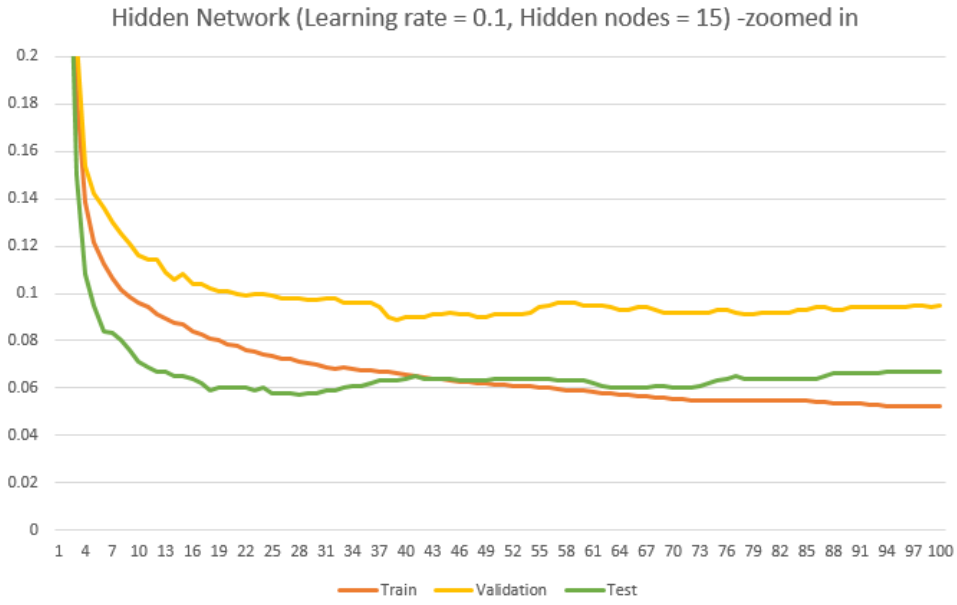
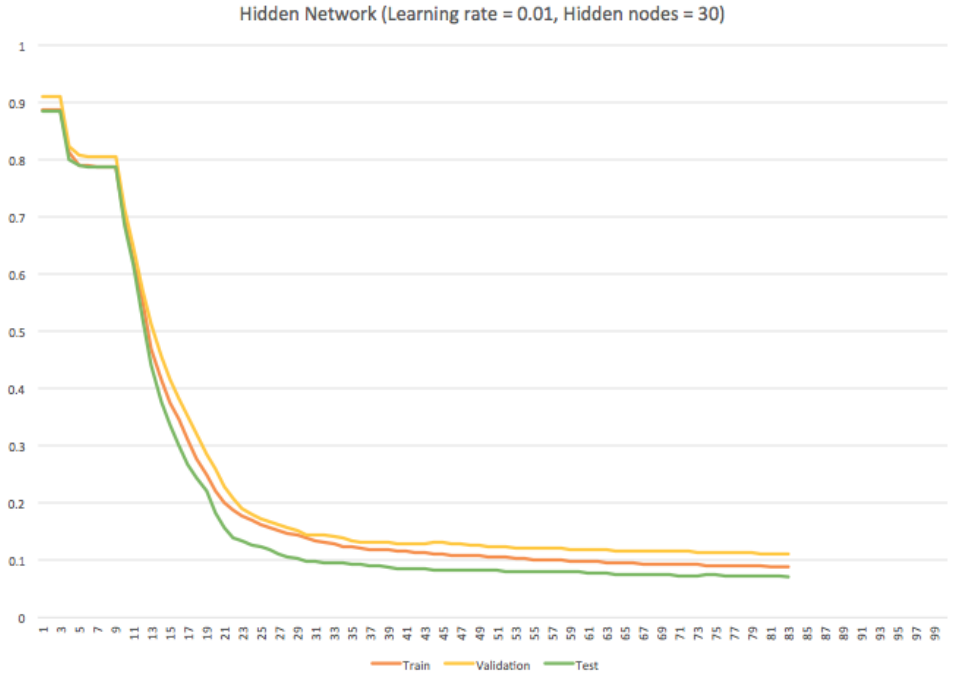


Chart of Training Set and Validation Set Error vs. Number of Epochs (30 hidden units)



d) For 15 hidden units with learning rate 1.0, we stopped after 10 epochs (2 consecutive decreases after). With the **optimal** learning rate 0.1, we stopped after 54 epochs. With learning rate 0.01, we stopped near 100 epochs (might need more than 100 epochs). With the optimal learning rate 0.001, we stopped near 100 epochs (might need more than 100 epochs).

For 30 hidden units with learning rate 1.0, we stopped after 6 epochs (2 consecutive decreases after). With the **optimal** learning rate 0.1, we stopped after 79 epochs. With

learning rate 0.01, we stopped after 83 epochs. With learning rate 0.001, we stopped after 83 epochs.

e) For 15 units, using optimal learning rate 0.1 and stopping after 54 epochs, we get train performance of 0.94, validation performance of 0.91, and test performance of 0.935.

For 30 units, using optimal learning rate 0.1 and stopping after 79 epochs, we get train performance of 0.97, validation performance of 0.93, and test performance of 0.961.

By comparing the two, we would choose the network structure of 30 hidden units given that it outperforms the 15 units network structure on all the training, validation and test sets.

f) The test set performance of our chosen network with 30 hidden units and an optimal rate of 0.1 is 0.961 as mentioned above. This does significantly better than the test set performance of the committee of perceptrons with learning rate 1.0 in 3.6 which hovered around 0.91.

(8) Hidden Network

a) We experimented with adding a second hidden layer with various numbers of hidden units. We ran experiments letting the two hidden layers have the same number of hidden units, ranging from a small number of hidden units to a larger number of hidden units. We also experimented with varying the number of hidden units in each layer, for example having 10 in the first and 15 in the second.

Having achieved success with one layer, we thought we could improve on this success by adding another hidden layer. We reasoned that the automatic feature engineering intuition associated with neural networks would result in the first hidden layer passing the second hidden layer an even better set of features which the second hidden layer can then do learning on.

b) The neural networks we trained with two hidden layers universally had very poor performance on training, validation, and test sets. For all of the experiments we ran, all had training error, validation set error, and test error at 0.12 or worse. The networks would converge within three or four epochs: the errors would stop changing, indicating that the weights stopped changing. After carefully spending time carefully validating that our network was indeed constructed in the correct fashion, we determined that these poor results were not the result of a bug, but because using two hidden layers instead of one with this particular domain had very poor performance in general.

The fact that the errors and weights stopped changing after a small amount of epochs indicate to us that the neural network with two hidden layers converges to a local minima extremely quickly. The fact that having two hidden layers converges rapidly to local minima that does barely better than chance indicates that one must be careful when adding additional hidden layers blindly.

Because, unlike LeCun et al, we did not do feature engineering with the extra hidden layers we had, there may not have been an actual benefit to having an extra hidden layer

because each hidden layer was assigned the exact same task. In fact, because each hidden layer was only given instructions to learn the general structure, they may have interfered with the function of the other, leading to vastly inferior results. Because our particular algorithm does not deal with escaping from and searching around local minima, combining that fact with the high susceptibility of two hidden layers hitting a local minima early on leads to poor results.

Having two hidden layers without special engineering may also be suspect because the second hidden layer works with values that are summed and then passed through the sigmoid function which changes the distribution of values. In addition, classification now depends on the weighted output of the second layer, which in turn depends on the weighted outputs of the first layer. With all nodes feeding in to each other this could create a situation where there is a lot of interplay or interference in between layers. From this experiment we conclude that without careful feature engineering, additional hidden layers should be added only with care.

4. Alternative Error Function

(1) The difference between the error function C and the loss function L is that the former implements some kind of regularization that penalizes large w_{km} and w_{mj} weights. Since we're trying to minimize C, we would also want to minimize the magnitude of w_{km} and w_{mj}.

The added penalty terms cause the weights to converge to smaller magnitudes than they otherwise would. Large weights can hurt generalization in a number of ways. Extremely large weights from input to hidden units and extremely large weights from hidden to output units can cause greater variance in outputs beyond the range of the data if the activation function does not have the same range as the input data. This error function C prevents the network from using weights that it does not need, from overfitting the noise in the data and from stretching the variance of outputs too much.

(2)

$$C(w) = \sum_{n=1}^{N}\sum_{j=1}^{J}(y_{nj} - a_{nj})^2 + \lambda \sum_{m=1}^{M}\left(\sum_{k=0}^{K} w_{km}^2 + \sum_{j=1}^{J} w_{mj}^2\right)$$

For updating weights w_{km} between input k and hidden input m, for a particular example, we have:

$$\frac{\partial C(w)}{\partial w_{km}} = \frac{\partial}{\partial w_{km}}\left(\sum_{j=1}^{J}(y_j - a_j)^2\right) + \lambda\frac{\partial}{\partial w_{km}}\left(\sum_{m'=1}^{M}\sum_{k'=0}^{K} w_{k'm'}^2\right) + \lambda\frac{\partial}{\partial w_{km}}\left(\sum_{m'=1}^{M}\sum_{j'=1}^{J} w_{m'j'}^2\right)$$

The first term on the right-hand side is exactly the same as that in the loss function. We've derived the partial derivative or the first term in the Lecture 6 notes on page 9.

The last term or partial derivative on the right-hand side of the equation is just 0 since w_{km}

doesn't appear in the expression $w^2_{m'j'}$. Thus, we only have to simplify the middle term. In the middle term, the weight w_{km} only influences the mth hidden unit, so only one element in the summation matters. Simplifying, we get the following gradient descent:

$$= -2g'(z_m)x_k \sum_{j=1}^{J} \delta_j w_{mj} + \lambda 2 w_{km}$$

$$\propto (a_k \delta_m + \lambda w_{km})$$

Deriving a weight update rule, we get $w^{(r+1)}_{km} \leftarrow w^{(r)}_{km} + \alpha \left(a_k \delta_m + \lambda w^{(r)}_{km} \right)$. Combining terms and constants, we have, $w^{(r+1)}_{km} \leftarrow (1 + \lambda') w^{(r)}_{km} + \alpha \left(a_k \delta_m \right)$.

For updating weights w_{mj} between hidden unit m and output j for a particular example, we take the partial derivatives with respect to w_{mj}, go through the same process and arrive at the following similar weight update rule: $w^{(r+1)}_{mj} \leftarrow (1 + \lambda') w^{(r)}_{mj} + \alpha \left(a_m \delta_j \right)$.