Joe Shymanski 5/1/21

Project 4: Supervised Learning

(Decision Trees & Feedforward, Multilayer Neural Networks)

Objective

The objective of this project is to implement a Decision Tree Learning (DTL) algorithm and a Feedforward, Multilayer Artificial Neural Network (ANN) with Backpropagation in order to test against several different classification datasets.

Implementation

DTL

The DTL algorithm is much simpler to understand and implement. It takes several inputs: the list of training examples along with their proper classifications, the list of attributes that all examples possess, the list of parent examples, the function to measure node impurity, the maximum depth, and the current depth. The algorithm will then enter a recursive loop with itself.

DTL always checks for a few terminating conditions at the beginning of each loop. If there are no more examples, it returns a node containing the class with the most instances in the parent examples. If there are some examples but they all possess the same classification, it returns a node with this classification. If the examples have different classes but either there are no more attributes or it has reached its maximum depth, DTL returns the most common class among the remaining examples. If none of these conditions are met, then it can begin to choose attributes from which to branch.

The algorithm will pass the remaining attributes, examples, and node impurity measure to the argmax function. This function will return the attribute (and split point, if the attribute is continuous) with the largest Information Gain. It does so by looping over all remaining attributes and passing the current attribute, the list of remaining examples, the list of remaining attributes, and node impurity measure to the importance function. This importance function calculates Information Gain using the specified impurity measure, either Entropy or Gini. If the attribute is discrete, impurity is measured on each value of the attribute. If it is continuous, split points are determined using a sorted list of examples whenever two consecutive examples have differing classes. Then impurity is calculated for both the "low" example set and the "high" example set, and the split point with lowest impurity is returned.

After argmax returns an attribute, a decision tree is created with the selected attribute as the root node. DTL then checks if a split point was returned, meaning the attribute has continuous values. If the attribute is discrete, the algorithm loops over all of the attribute's values. With each pass, a list is created of examples with that value of the attribute, and a subtree is created by passing the example subset, the list of attributes minus the selected one, the example superset as the new list of parent examples, and the depth increased by one. If the attribute is continuous, the same is done with the examples below the split point and those above the split point, except the selected attribute is not removed when passing the attributes as an argument. After each subtree is returned, it is added as a branch to the tree with the branch label as the value of the root attribute and the child node as the subtree. Then the tree is returned.

ANN

The ANN learner is a bit more difficult. It accepts the list of training examples with their proper classifications, the network, and an alpha value, also known as the learning rate. It is important to understand the architecture of the network before understanding the algorithm itself.

The network object contains four attributes: a list of layers, a matrix of weights, an activation function, and the derivative of that activation function. It accepts the list of layers and a standard deviation in its constructor and builds the weight matrix from these layers, initializing each cell to a random value according to a normal distribution with a mean of 0 and the specified standard deviation. The network could accept other activation functions, but the sigmoid function was the only one used for this project.

The structure of the weight matrix is also a bit difficult to understand. The first index specifies the “leftmost” layer to which the weights belong. The second index specifies the node in the “right” layer, and the third index specifies the node in the “left” layer. Thus, the dimensions of the weight matrix are $N - 1 \times L_R \times L_L$, where N is the number of layers, L_R is the number of nodes in the right layer, and L_L is the number of nodes in the left layer.

Now that the network structure has been created, the algorithm can begin. First, the algorithm is wrapping in a loop which iterates an arbitrary number of times. Within each outer loop, the algorithm again loops over each example. To start, the ANN learner simply propagates, or feeds forward, the inputs from the input layer all the way to the output layer. The initial layer simply stores the input values. For every subsequent layer, it loops over all the nodes of the current layer, which is the right layer. For each of those nodes, it loops over all the nodes in the previous layer, which is the left layer, and sums up the products of the weight of each connection and the output value from the left layer’s node. The sum, known as the right node’s input value, is stored, and then it is fed into the activation function. This result is stored as the right node’s output.

After this, the algorithm then propagates backward. It starts by calculating and storing the errors of every node in the output layer. This is done by multiplying the activation function’s derivate of the node’s input value with the difference between the target output value and the calculated output value. For each subsequent layer going backwards, the weights are updated using these errors. The product of the weight between the left and right nodes and the error of the right node is calculated. These products are summed up over all nodes in the right layer, and this sum is multiplied by the activation function's derivative of the left node's input.

Finally, once backpropagation has finished, all of the weights in the network's weight matrix are updated. This is done by adding the product of alpha, the output at each left node, and the error at each right node. Once all examples have been considered, the algorithm loops the specified number of times, then returns the network object.

Results & Analysis

DTL

Figure 1

Example of Decision Tree on Continuous Data

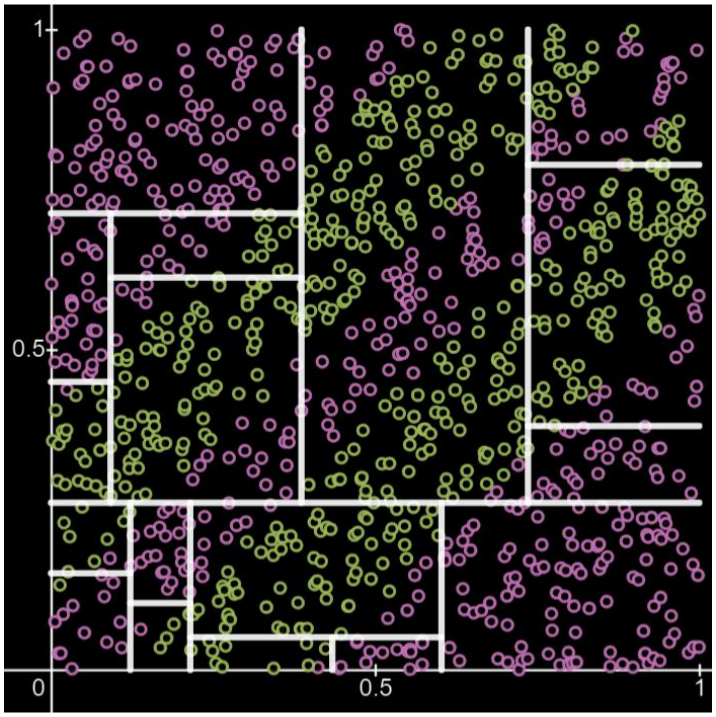
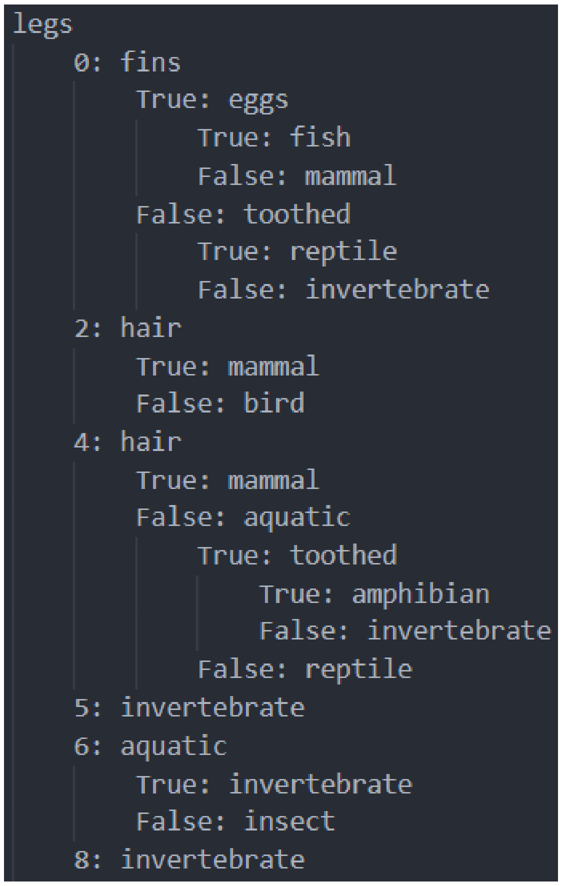


Figure 1 shows an example of the boundaries made by DTL over a dataset with two classifications and two attributes, both of which are continuous. The resulting decision tree was limited to a maximum depth of 5 in order to avoid overfitting and infinite recursion. It is clear that univariate decision trees often underperform for continuous datasets with boundaries that are not parallel to the axes.

Figure 2

Example of Decision Tree on Discrete Data



```
legs
    0: fins
        True: eggs
            True: fish
            False: mammal
        False: toothed
            True: reptile
            False: invertebrate
    2: hair
        True: mammal
        False: bird
    4: hair
        True: mammal
        False: aquatic
            True: toothed
                True: amphibian
                False: invertebrate
            False: reptile
    5: invertebrate
    6: aquatic
        True: invertebrate
        False: insect
    8: invertebrate
```

Figure 2 shows how a decision tree is formed on a dataset whose attributes only have discrete values. This is the decision tree for the entire dataset of zoo animals from the UCI database, and it is relatively simple. There are seven different classes that each animal falls under. This tree was also limited to a maximum depth of 5 when it was created, but it ended on its own before then. Trees with mostly discrete attributes and almost no noise usually form quickly and are relatively shallow.

Table 1

Decision Tree Accuracies Using 5-Fold Cross-Validation

Dataset	Training Set Accuracy (%)	Testing Set Accuracy (%)
Project 1	82.03	74.40
Zoo	100.00	97.05
Letter Recognition	55.10	38.60

Note. All trees used maximum depth of 5 and Entropy as their impurity measure.

Table 1 demonstrates the various training set and testing set accuracies across three different datasets using 5-fold cross-validation. All used Entropy as their node impurity measure for the sake of consistency. In fact, this algorithm has yet to create a decision tree differently based on which impurity measure was used, even though the Information Gain values are not exactly the same between the two measures. Thus, it seems the two measures may be directly related and output the same relative magnitudes for attributes in the argmax function.

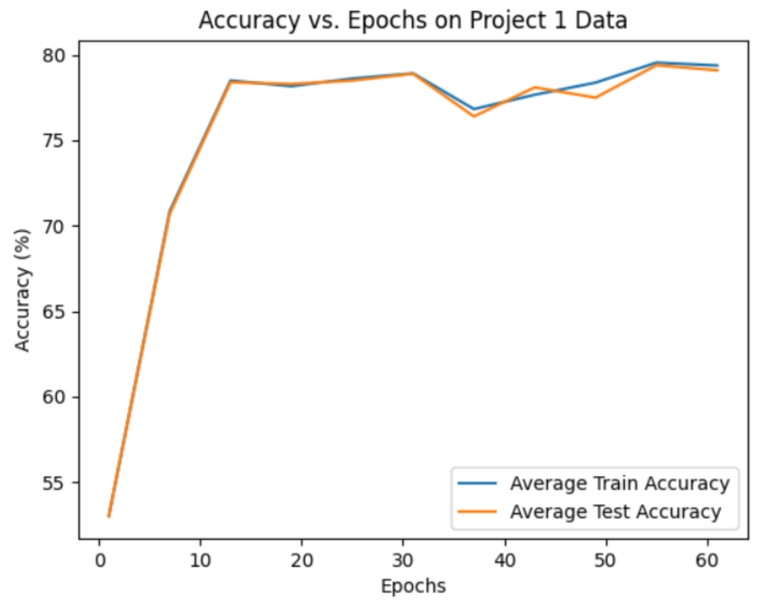
DTL performed the best by far on the Zoo dataset. This is likely due to the fact that all the attributes in this dataset are discrete, and they closely follow the attributes that biologists generally to make the classifications in the first place. There was little difference between training set and testing set accuracy, suggesting little to no overfitting on this dataset.

The algorithm performed a bit worse on the Project 1 dataset. The reason for this, which has been discussed once before, is due to the class boundaries not being parallel to the attribute axes. The maximum depth cutoff may also throw off the accuracy slightly since it is a suboptimal strategy to prevent overfitting. Post-pruning would likely yield better accuracy. The accuracy drops a bit between training and testing sets, suggesting the trees may still be slightly overfit to the training sets.

Finally, it appears DTL performed pitifully on the Letter Recognition dataset, but this is not exactly the case. This dataset features 26 distinct classifications – one for every capital letter in the alphabet – so the algorithm actually does much better than the null model of guessing either randomly or the same letter each time, which would yield about a 4% accuracy. On top of that, the dataset contains about 20,000 entries, so the trees only worked with a single random subset of 500 entries. Of the three datasets learned, the Letter Recognition dataset certainly appears to lead to the most overfitting.

ANN

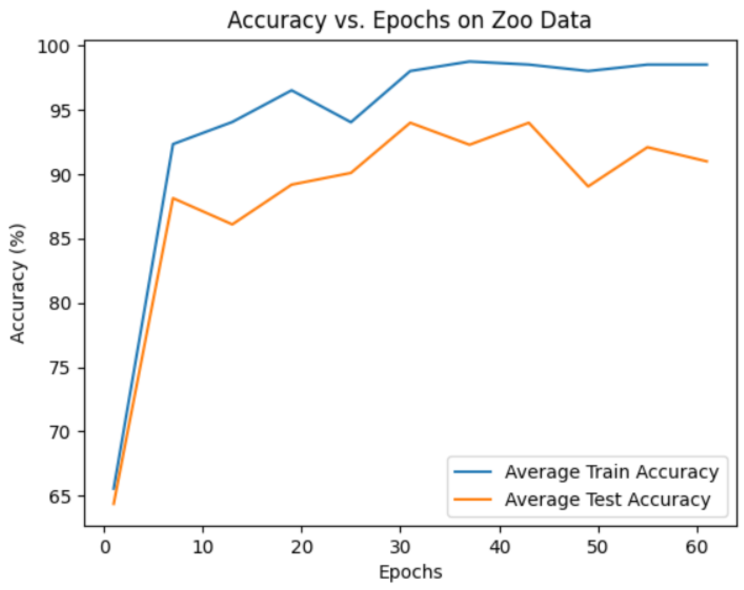
Figure 3



Note. ANNs built using 1 hidden layer with 10 nodes, standard deviation of 1, and learning rate of 1.

Figures 3, 4, and 5 demonstrate the relationship between average training set and testing set accuracies and the number of epochs used to build each ANN. Each data point was calculated using 5-fold cross-validation. As you can see in Figure 3, the ANNs for the data from Project 1 demonstrate little to no overfitting since there is no significant gap between the train accuracies and the test accuracies. The ANNs for the Zoo data, however, demonstrate a relatively significant amount of overfitting. As the number of epochs used to generate the ANNs increases, the level of overfitting also roughly increases, as is shown in Figure 4. Figure 5 shows that there is perhaps a slight amount of overfitting with the ANNs for the Letter Recognition data, which gets marginally more significant as the number of epochs increases.

Figure 4

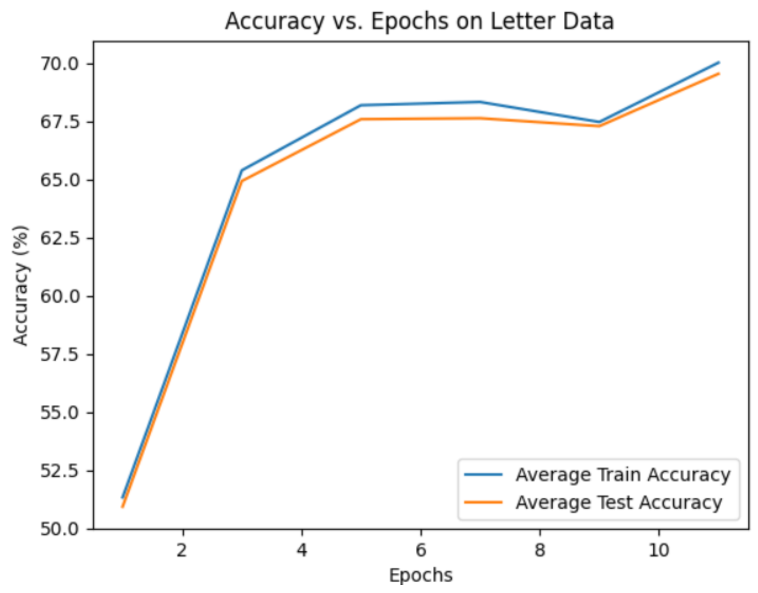


Note. ANNs built using 1 hidden layer with 20 nodes, standard deviation of 1, and learning rate of 1.

The overfitting may arise from a number of factors. Perhaps smaller datasets show significant overfitting more quickly than larger ones. Maybe ANNs with a large number of input or output nodes are more susceptible for overfitting. Whatever the case, it is important to be aware of overfitting when using any of these ANNs to make predictions on unseen data.

Besides overfitting, the figures show a clear positive relationship between epochs and training set accuracy. Most of the testing set accuracies also demonstrate this, except for the Zoo data, which shows test accuracy may have been decreasing as the epochs grew, due to overfitting. The relationship appears to be logarithmic, meaning the rate of increase in accuracy is decreasing as number of epochs increases.

Figure 5

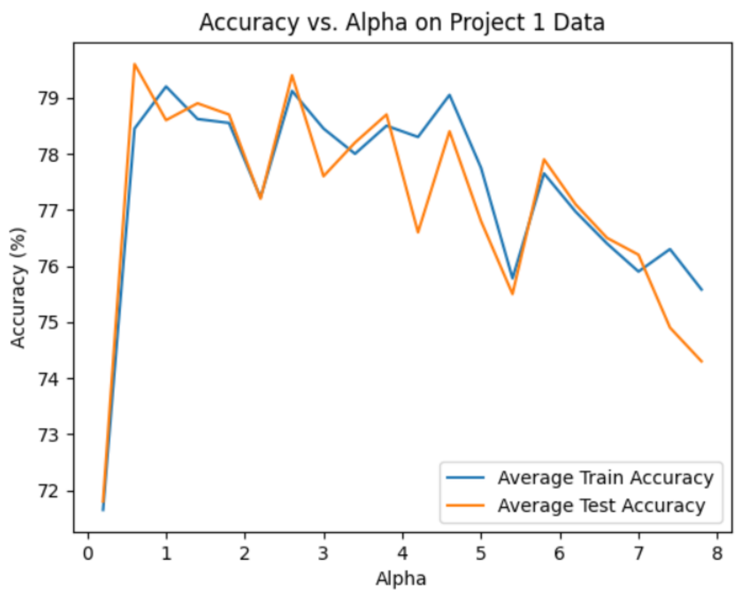


Note. ANNs built using 1 hidden layer with 10 nodes, standard deviation of 1, and learning rate of 1.

The time it takes to perform 5-fold cross-validation for ANNs is directly related to the number of layers, the number of nodes in each layer, the number of epochs, the number of examples in the training and testing sets, and the number of inputs and outputs. To serve as an example, the ANNs created using 11 epochs in Figure 5 took about 160 seconds each to form and predict the corresponding test data. This means that single data point took over 13 minutes to compute. In order to save time, at the direct expense of accuracy, I will be using the Project 1 dataset with relatively low numbers of epochs and nodes in its only hidden layer for further analysis. This is due to the fact that the Project 1 dataset only has 1,000 data points, compared to the 20,000 in the Letter Recognition data set, and its ANNs exhibited almost no overfitting.

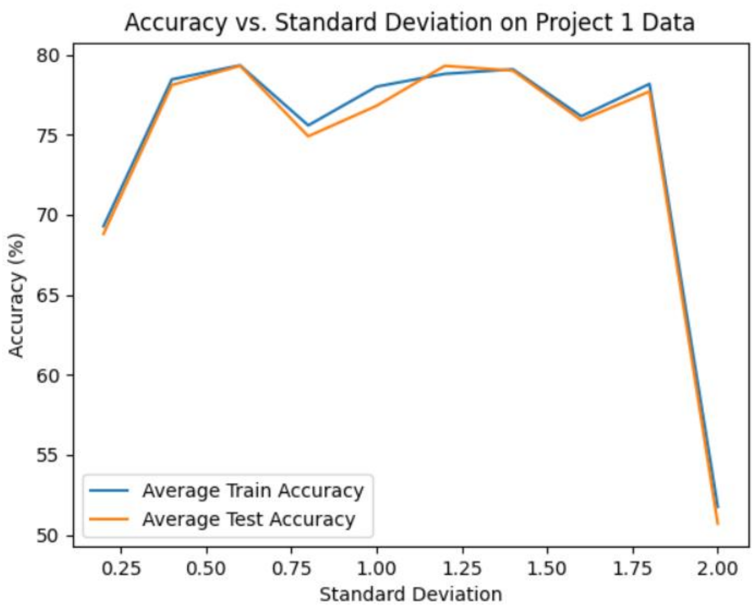
As you can see in Figure 6, accuracy initially spikes around a learning rate (alpha) value of 1 and begins to slowly decrease as it increases. Thus, it seems exceptionally high learning rates are not as effective on this dataset, but neither are exceptionally low learning rates. Somewhere between 0.5 and 4 seems to be a reasonable value for the learning rate.

Figure 6



Note. ANNs built using 1 hidden layer with 10 nodes, 20 epochs, and standard deviation of 1.

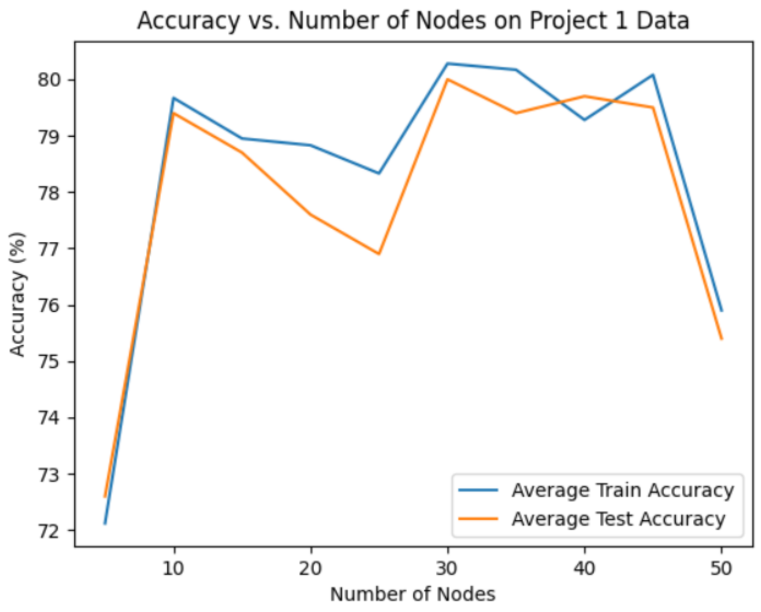
Figure 7



Note. ANNs built using 1 hidden layer with 10 nodes, 20 epochs, and learning rate of 1.

Figure 7 shows the relationship between accuracy and the standard deviation for the normal distribution that generates the initial random weights. Very small standard deviations (less than 0.25) and very large standard deviations (greater than 1.75) lead to diminished ANN performance. The standard deviations that lie somewhere in between these extremes all performed well and predicted at about the same level of accuracy. Thus, any standard deviation between 0.25 and 1.75 seems to be a reasonable choice for an accurate ANN.

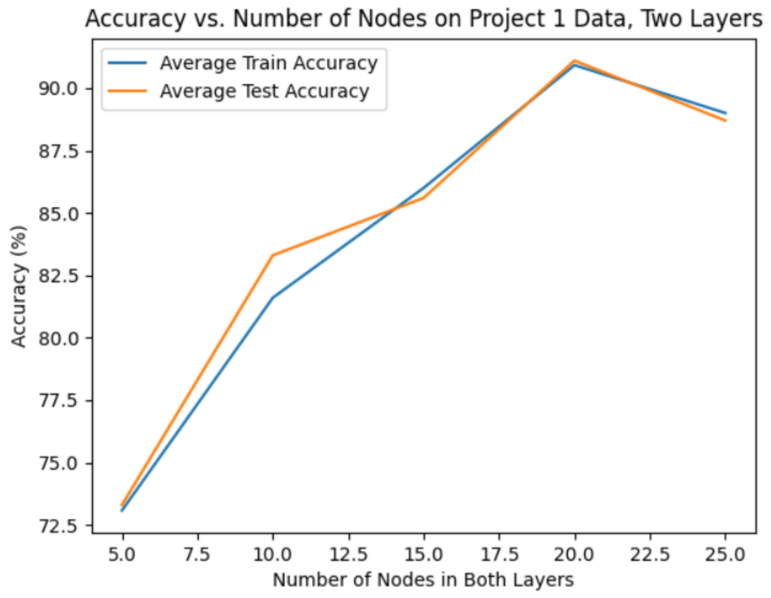
Figure 8



Note. ANNs built using 1 hidden layer, 20 epochs, standard deviation of 1, and learning rate of 1.

The relationship between accuracy and the number of nodes in a single hidden layer, displayed in Figure 8, looks quite similar to Figure 7. For low numbers of nodes (less than 10) and high numbers of nodes (more than 45), accuracy decreases pretty significantly. In between the values, the accuracy fluctuates a bit, with peaks and valleys that may or may not be significant. In addition to these fluctuations, the level of overfitting seems to fluctuate. ANNs with 20 and 25 hidden layer nodes seem to be fairly overfit while the ANNs with 40 nodes do not seem to be overfit. The significance of these fluctuations is also unknown.

Figure 9



Note. ANNs built using 2 hidden layers with equal number of nodes, 20 epochs, standard deviation of 1, and learning rate of 1.

As you can see in Figure 9, accuracy seems to increase as the number of nodes in both hidden layers of the network increase. This increase may peak around 20 nodes each and drop off after that, it may plateau and drop off a bit later, as was the case in Figure 8, or it may continually increase. I believe the second scenario is most likely. The accuracies in this scenario are markedly higher, on average, than those in the single hidden layer scenario.

With all of this in mind, it seems an ANN with two hidden layers with about 20 nodes in each layer, a high number of epochs, say, 50, a standard deviation of 1, and a learning rate of 1 should produce the most accuracy predictions. After performing 5-fold cross-validation on such ANNs for the Project 1 dataset, they produced average training and testing set accuracies of 91.5% and 91.0%, respectively, and each took about 40.88 seconds to form and predict. For the Zoo data, the accuracies were 97.28% and 90.05%, each ANN taking 7.29 seconds on average.

The Letter Recognition data, using 5 epochs instead of 50, produced accuracies of 72.49% and 71.78%, averaging 201.15 seconds per ANN. These are summarized in Table 2.

Table 2

ANN Accuracies Using 5-Fold Cross-Validation

Dataset	Training Set Accuracy (%)	Testing Set Accuracy (%)	Time Elapsed (s)
Project 1	91.50	91.00	40.88
Zoo	97.28	90.05	7.29
Letters	72.49	71.78	201.15

Conclusion

These supervised learning algorithms work very well for certain domains. Decision trees work great for datasets with discrete attributes where the classifications follow some underlying decision process. ANNs are good for datasets with continuous attributes, something that decision trees can sometimes struggle with. They also lead into the deep learning field of computer science, so they are clearly useful for a plethora of applications. The DTL in this project created fairly accurate decision trees in almost no time. It slightly overfit some trees more than others, depending on the dataset. The ANN learner took a bit longer to produce its most accurate networks for each dataset, but generally produced much better predictions. Some ANNs were more overfit than others, depending on the dataset, but the overfitting was generally less drastic than the decision trees.