Jonathan Klinger
Implementation of Multilayer Perceptron
Project 5

Abstract:
The Multilayer Perceptron (MLP) is a very strong machine learning algorithm in the neural network family. As opposed to the single layer perceptron, MLP has 1+ hidden layers that enables predictions on non-linear problems. In this project, MLP was used to run predictions on 5 datasets. The Breast Cancer and House Votes datasets were predicted using binary classification. The Iris, Soybean, and Glass datasets were predicted using multi-class classification. Prediction accuracy was very strong for MLP, with 97%+ performance on the breast cancer, house votes, and iris datasets. It was also able to achieve 100% accuracy on the soybean dataset. However, the results on the glass dataset were not as strong, peaking at an accuracy a little bit above 64%.

Introduction:
My hypotheses going into this project were as follows:

1. Using less complex machine learning models (such as logistic regression and adaline), we were able to achieve very high performance one 4/5 datasets. For these datasets, adding too much complexity (adding too many hidden layers and adding too many hidden units) may lead to the MLP overfitting the data and lower prediction accuracy.
2. While previous models may show weak results on the glass dataset, I would expect the MLP dataset to show stronger results. This is because the MLP is a universal approximater and is better suited to learning more complex datasets.

Implementation of MLP:
My implementation of MLP takes the batch approach.
The fit method begins by feature scaling the training data using normalization or standardization (default = no scaling). The weights are then initialized based off of the inputted layers with shape (previous layer, current layer) (previous layer = feature count for the first hidden layer).
The training data is then forward propagated through the network to perform initial predictions based off of the initialized weights. To do this we use the following formulas:

$$output_l = sigmoid((units_l \cdot weights_l) + bias_l)$$

Formula 1: output of each layer

$$sigmoid(x) = \frac{1}{1 + e^x}$$

Formula 2: sigmoid equation

In Formula 1, *l* stands for the current layer and units are the outputs of each node for the previous layer (training sample for first hidden layer). The prediction/output follows the same equation for the output layer for binary classification. For multi-class classification the softmax function (Figure 3 and 4) is used instead of the sigmoid function.

$$output_l = softmax((units_l \cdot weights_l) + bias_l)$$

Formula 3: output of the output layer

$$softmax(x) = \frac{e^x}{\Sigma e^x}$$

Formula 4: softmax equation

Once the initial predictions are computed, we backpropagate the errors through the network using gradient descent. Starting with the weight updates for the output layer, we calculate the error, multiply the errors by the hidden units and the learning rate (Formula 6). Using the chain rule the error is passed down to the next layer (Formula 5).

$$error = (y' - y)$$

Formula 5: error of the final layer

$$weightupdates_l = (y' - y) \cdot units_l^T$$

Formula 6: weight updates for the final layer

Thus the next layer will have the weight update that includes the error and the weights of the output layer (Formula 7), where *sigmoid'* represents the derivative of the sigmoid value (Formula 8).

$$weightupdates_l = ((y' - y) \cdot weights_{l+1}) * sigmoid'(inputs_l)) \cdot units_l^T$$

Formula 7: weight updates for intermediate hidden layer before the output layer

$$sigmoid(x)' = sigmoid(x) * (1 - sigmoid(x))$$

Formula 8: sigmoid derivative function

The error passed to the next layer down is shown in Formula 9, and replaced with the previous error in the previous weight update equation, for the new weight update (Formula 10).

$$error = ((y' - y) \cdot weights_{l+1}) * sigmoid'(inputs_l))$$

Formula 9: error for intermediate hidden layer 2 before the output layer

$$weightupdates_l = ((error \cdot weights_{l+1}) * sigmoid'(inputs_l)) \cdot units_l^T$$

Formula 10: weight update for intermediate hidden layer 2 before the output layer

The bias updates for each layer are computed by simply by summing the errors. The weights are then subtracted by the updates*learning_rate, to update the weights and conclude backpropagation. We repeat the above process for the specified number of epochs.

When running predictions on the trained model, the test samples are forward propagated through the network and the output is computed. For binary classification, anything above 0.5 sigmoid score is predicted as 1, else 0. For multi-class classification, the argmax of the softmax output is taken as the prediction.

Experimental approach:

Hidden Layers and Neurons

While there are many approaches discussed and researched to defining the optimal number of hidden layers and units, each approach has their own drawback. For example, the dynamic node creation approach only works when we make the assumption of one hidden layer. The cascade correlation approach adds layers, but does not add nodes. On my first attempt at trying to devise a solution, I decided to take an alternative approach to the dynamic node creation approach. I started with one hidden layer and one node in the layer and found the most optimal number of nodes, assuming this number of nodes for the first hidden layer, I added an additional hidden layer and found the optimal number of nodes for that layer given the previous layer. However, I quickly realized that making this assumption for the first layer added bias to my optimal 2 hidden layer network. I did some research on alternative approaches to optimizing the number of hidden layers and units, but most research seemed to be inconclusive. Karsoliya sums up this point very well on just optimizing hidden nodes, “many researcher put their best effort in analyzing the solution to the problem that how many neurons are kept in hidden layer in order to get the best result, but unfortunately no body succeed in finding the optimal formula for calculating the number of neurons that should be kept in the hidden layer so that the neural network training time can be reduced and also accuracy in determining target output can be increased.”[1]

For this reason, I decided to use a more brute force approach to optimizing my hidden layers and units. In Behaviour Analysis of Multilayer Perceptrons with Multiple Hidden Neurons and Hidden Layers, Panchal discusses a few rules of thumb when choosing the optimal number of hidden units for a 1 hidden layer network: 1) the number of hidden neurons should be between

the size of the input layer and the size of the output layer; 2) the number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer; and 3) the number of hidden neurons should be less than twice the size of the input layer. Both Panchal and Karsoliya highlight that these rules are not fool proof and Panchal says that they are more of a starting point. My approach was to stretch these rules when testing, providing hidden unit sizes above and below the limits of these rules. If I found that my peak performance was nearing the edge of the higher or lower end of my range, I would expand my testing in that direction. I did the same for the number of hidden layers, starting from the minimum amounts of 1 or 2 hidden layers and expanding if I see that more hidden layers were improving performance.

All of these defined hidden units and layers were tested on all possible combinations with a range of learning rates and epochs. While this proved to be a very computationally expensive task, it did help me identify the optimal hyper parameters with high confidence. However, I know that in certain circumstances, the time to test an array of hyperparameters would render the approach intractable.

Training approach
Originally, my training approach involved running the entire training sample forward and backward through the network for the number of epochs. I found two issues with this method: during each training epoch, 1) I was showing the same exact data to the network; and 2) the data was in the same order. In Deep Imitation learning with memory for Robocup Soccer Simulation, Hussein mentions that “keeping the sequence of samples without utilizing memory can be detrimental to training as the samples in training batches will be too similar and lack diversity.” [2]

Additionally, I decided to use the batch approach over the on-line approach, because in Theoretical analysis of batch and on-line training for gradient descentlearning in neural networks, Nakama concludes that the batch approach improves convergence when the loss function is quadratic and I decided to use Mean Squared Error as my loss function.

Momentum
I provided the option for using momentum. It can be used for speeding up time to convergence. This option was used to see if there would be any improvement if the required convergence time was much longer without using momentum.

Hyperparameter tuning
Hidden layers, number of neurons, learning rate, epochs, and momentum were all optimized by testing a number of values. The values are accepted in list form and all permutations of all the lists were tested to see which set of hyperparameters were most optimal.

Results:

	Hidden Layers	Learning rate	Epochs	Accuracy
Breast cancer	[9]	0.01	250	97.4%
Iris	[30]	0.001	500	98.6%**
House votes	[3]	0.01	750	97%
Glass	[30]	0.01	500	64.1%*
Soybean	[5]	0.01	250	100%

Table 1: results on each dataset with optimal hyperparameters (“” denotes using momentum of 0.75 and 0.6 for “**”)*

In every case, the optimal number of hidden layers was 1. For the breast cancer, iris, house votes, and soybean dataset, adding a second layer dramatically decreased performance (30-60%). For the Glas dataset, adding a second hidden layer yielded similar but slightly lower results.

Behavior of algorithm:
Overall the performance of the Multilayer Perceptron was very strong for 4/5 datasets. It was able to perform on par or better than logistic regression (which was the clear outperformer on the same datasets from the last project). However, we didn't see too much improvement on the high scoring datasets. One might first assume that this is because logistic regression performed almost perfectly on these datasets, so there is not much room to improve. While this is true, it goes even deeper. We didn't see much improvement on these datasets because the results using logistic regression showed that the classes in these datasets were linearly separable. To prove this, I reduced the dimensions of the datasets using PCA with two principal components and plotted the reduced data in two dimensions. In Figure 1 and 2 I show the plots of the Iris and soybean datasets. These two datasets were multiclass problems like the glass dataset but seem linearly separable as opposed to the plot of the glass dataset in figure 3.

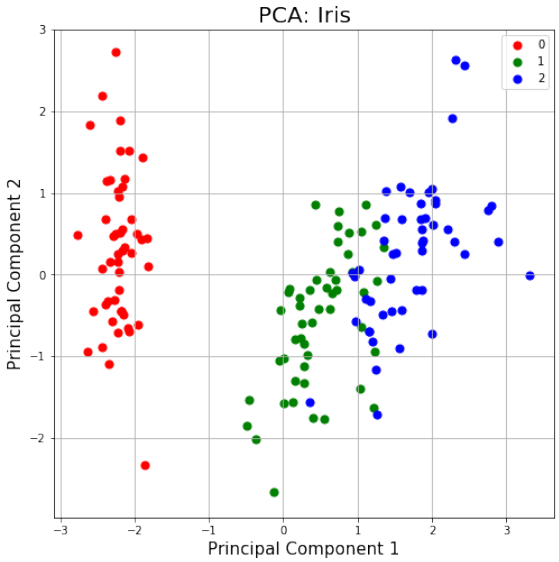

Figure 1: PCA for the Iris dataset (0,1,2 = different classes)

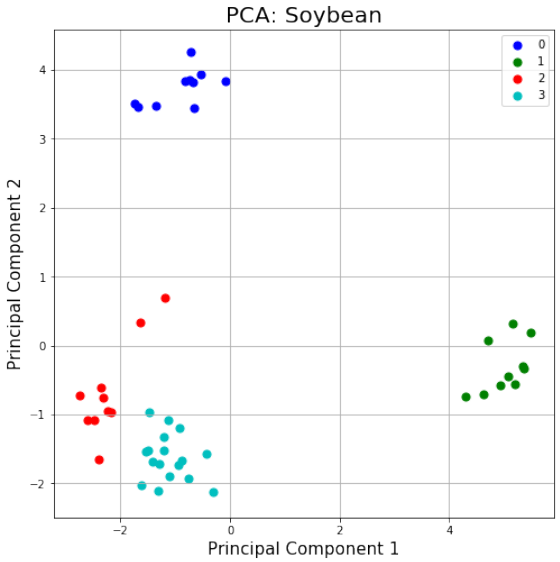

Figure 2: PCA for the Soybean dataset (0,1,2,3 = different classes)

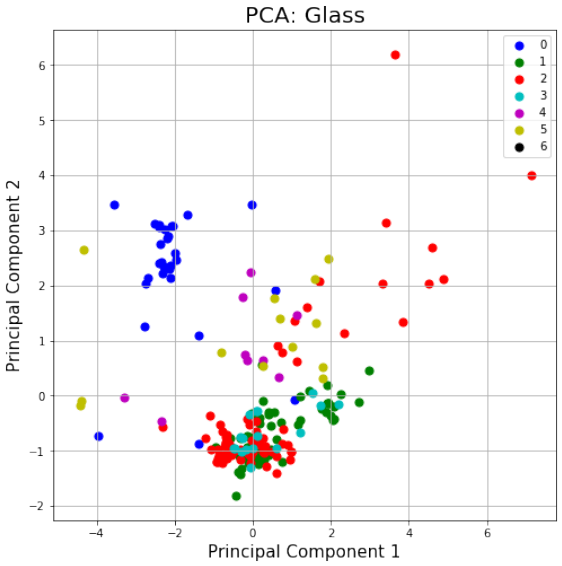

Figure 3: PCA for the Glass dataset (0,1,2,3,4,5,6 = different classes)

Going into this project, I assumed that there would be minute (if not 0) improvement on these datasets because we had already proved linear separability, and I was correct. Thus, using an MLP on these datasets was a bit overkill for the given problem. This conclusion also confirmed another one of my hypotheses, which was that achieving optimal results on these datasets would only require 1 hidden layer. Increasing the number of layers led to a dramatic decrease in performance in most cases (around 30-60%), while increasing the number of nodes significantly also lead to a decrease in performance. This was most likely due to the model overfitting the training data due to the increased level of abstraction. In Feedforward Neural Networks with a Hidden Layer Regularization Method, Alemu notes that “having too many neurons in the hidden layer may lead to overfitting of the data and poor generalization while having too few neurons in the hidden layer may not provide a network that learns the data.”[3] This confirmed my overfitting theory.

One thing I found extremely interesting that I could not find any research discussing is that my MLP was more sensitive to the addition of a single hidden layer than it was to a reasonable increase in neurons for the hidden layer. Most papers I found discuss too many neurons in the hidden layer leading to a decrease in performance. This was true when I dramatically increased the number of neurons but not when I increased it by 20-30. However, in On the Expressive Power of Deep Neural Networks, Raghu concludes “our analysis of trajectories provides insights for the performance of trained networks as well, suggesting that networks in practice may be more sensitive to small perturbations in weights at lower layers.”[3] Given that our neural network was not to deep, this might have been why we saw that the number of neurons were not as sensitive as the number of layers.

While most of the results weren’t particularly exciting, the glass dataset was the saving grace. Using an MLP improved the prediction accuracy on this dataset by over 8%. This was a nice sized jump and proved that the dataset can benefit with the increased complexity of the model and universal approximation that comes with MLP. While 64.1% accuracy does not seem like much, it is a 6 class classification problem with only a little over 200 samples. I also found that this dataset was less sensitive to additional layers, which means that the abstraction helped improve prediction, but it was interesting to see that accuracy did not increase with an additional layer. While it is a popular opinion that rarely is a second hidden layer required, I did some additional research and found a paper Backpropagation Neural Nets with One and Two

Hidden Layers, where de Villiers concludes that in their research “there is no statistically significant difference between the optimal performance of three and four-layered nets” [5] (1 and 2 hidden layers) and that 1 hidden layer actually performs better on average. This was exactly what I was seeing in the results for the glass dataset.

Because we have proved that the glass dataset is not linearly separable and that we need a more complex algorithm, I would assume that this problem would require more training data to yield higher accuracy. I did some digging and found a NASA paper called Training data requirement for a neural network to predict aerodynamic coefficients, where Rajkumar states “When a subclass lies near a decision boundary, it is important to use a large dataset in order to avoid learning patterns of noise, which are in common among a large fraction of the representatives of that subclass. In general, proper selection of the training dataset leads to the success of the neural network classification or prediction. Larger networks require large training datasets. Over fitting is more likely when the model is large.”[6] This supports all my findings for the glass dataset. Firstly, a PCA visualization shows that there is virtually no boundaries between certain classes, so even more so we would need more training data. Additionally, the decreased performance from adding an additional layer could have been caused by overfitting, which could be remediated with more training data.

Conclusion:
At the very beginning of the course we learned about Ocaam’s Razor, which states that a simple approach is usually the best approach. In the last project I used logistic regression on the same 5 datasets and achieved 97%+ accuracy on 4 of the 5 datasets. This was evidence to me that these were linearly separable problems and using PCA, I was able to strengthen this hypothesis. While the optimal results for MLP on these datasets were just as good (or minutely better), it was as if we were ramming through the door when we could have used the key. This was evident when the optimal hyperparameters for each of these datasets had one hidden layer, relatively few neurons/units and any increase in hidden layers and neurons would dramatically decrease performance. On the other hand, MLP was able to noticeably improve accuracy on the glass dataset, which proves the value of its use for more complex problems. My conclusion in practice is thus the same as my conclusion in theory: when solving a machine learning problem, assess the complexity of the problem and choose the model best suited to tackle it. If need be, try simpler algorithms (like logistic regression) first, and try more and more complex models until you find that more complex models (than your current optimal model) do not improve results.

Citations
[1] Karsoliya, S., & Azad, M. (2012). Approximating Number of Hidden layer neurons in Multiple Hidden Layer BPNN Architecture.
[2] Hussein, A., Elyan, E., & Jayne, C. (2018). Deep Imitation Learning with Memory for Robocup Soccer Simulation. *EANN*.
[3] Alemu, H.Z., Wu, W., & Zhao, J. (2018). Feedforward Neural Networks with a Hidden Layer Regularization Method. *Symmetry, 10*, 525.
[4] Raghu, M., Poole, B., Kleinberg, J.M., Ganguli, S., & Sohl-Dickstein, J. (2017). On the Expressive Power of Deep Neural Networks. *ArXiv, abs/1606.05336*.
[5] Villiers, Jacques & Barnard, Etienne. (1993). Backpropagation Neural Nets with One and Two Hidden Layers. IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council. 4. 136-41. 10.1109/72.182704.
[6] Thirumalainambi, R., & Bardina, J. (2003). Training data requirement for a neural network to predict aerodynamic coefficients. *SPIE Defense + Commercial Sensing*.