Kordel France
Project 4
Class 605.649 Section 84
14 November 2021

Project 4 Analysis Document

FranceLab4 is a machine learning toolkit that implements several algorithms for classification and regression tasks. Specifically, the toolkit coordinates a linear network, a logistic regressor, an autoencoder, and a neural network that implements backpropagation; it also leverages data structures built in the preceding labs. *FranceLab4* is a software module written in Python 3.7 that facilitates such algorithms.

Introduction

Neural networks have proven very successful in constructing effective solutions to non-linear problems. They are able to build a knowledge representation about complex datasets that aid in classification, regression, and even reconstruction tasks. Backpropagation is a gradient-based algorithm that efficiently computes gradient losses and updates the network with knowledge gained from those losses. Also known as "Backprop", this algorithm allows a neural network to learn from erroneous decisions it makes about the target dataset such that, given any error $y_i - y_i^*$ for a specific data point *i,* an observation y_i , and a predicted value y_i^*, the error is corrected and weights associated with the incorrect decision are updated within the algorithm. This gives the neural network more comprehension over the data pushes it to converge the loss $y_i - y_i^*$ closer to zero. A well-constructed neural network can provide very high accuracy on complex datasets due to the incorporation of weight backpropagation.

Additionally, linear classifier and logistic regression algorithms are implemented as a separate task. Results for these two algorithms are also evaluated and compared in a similar manner as the neural network and backpropagation algorithms above. While simpler in nature than a neural network, both methods are effective both individually and as part of a larger dimensionality reduction algorithm.

In the following paragraphs, I illustrate the performance of a logistic regressor, a linear network, a neural network, and an autoencoder analyzing how back propagation effects the performance of the latter two algorithms. I show how different tuning parameters can find an optimal algorithm under each method and how the performance of the classification or regression task is affected as a consequence.

Hypothesis

There are four key goals of this project, so the construction of four hypotheses are warranted in order to evaluate the efficacy of the associated code.

Hypothesis 1: *"Given a dataset, FranceLab4 shall prove the efficacy of a logistic regressor in performing regression tasks while learning a set of optimized hyper parameters (the tuning values)."*

Hypothesis 2: *"Given a dataset, FranceLab4 shall prove the efficacy of a linear network in performing classification tasks while learning a set of optimized hyper parameters (the tuning values)."*

Hypothesis 3: *"Given a dataset, FranceLab4 shall prove that a neural network with layers previously trained by an autoencoder can provide better performance in at least one aspect of training."*

Hypothesis 4: *"Given a neural network, FranceLab4 shall prove the efficiency of Backpropagation in providing faster convergence toward a global minimum loss during learning."*

Results shall be measured in percentage of accuracy and mean squared error for classification and regression tasks, respectively. If the results presented in this paper are successful, we shall accept the null hypotheses; otherwise, we shall reject it. Throughout this paper, unless otherwise specified, I maintain Gaussian assumptions about the data until proven otherwise for reasons I elaborate on in the *Results* section.

Approach and Methodology

While there are four hypotheses, there are five evaluation points for this assignment - the algorithmic implementation each of *a logistic regressor, a linear network, an autoencoder, a neural network,* and *Backpropagation.* The *main.py* file contains all of the driver code for constructing the processing pipeline, running each of the five algorithms, and tuning a pruning value I denote as *alpha.* See the *README* file for more information on how the code is structured.

Data

The response variable for each trace run was essentially the last column in each of the six datasets, with preceding columns denoting the explanatory variables. As in my previous projects, I also included the ability to change the response variable. For example, when assessing the algorithm over the *Machine* dataset, I assessed the ability of *estimated relative performance* to be predicted by the preceding nine columns or a subset thereof; then I assessed the ability of the algorithm to predict *published relative performance* by the other nine columns. In *Project 3,* I continued this process until every column had been leveraged as a response variable. I have leveraged the knowledge gained fIn consideration of Table 1 and a data run for each response variable in each dataset, 2412 datasets were evaluated for each level of *k.*

Logistic Regression

A logistic regressor can be defined by the formula:

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_m X_m \qquad \textbf{Equation 1}$$

Where p denotes the probability of observing a value from the dataset, X_i denotes the i^{th} feature from m features of the data, and β_i denotes the weight associated with that feature.

Table 1 shows the weights computed by the regressor. The logistic regressor performed best on datasets that contained more continuous random variables. Discrete non-numeric variables were nominally encoded and represented as binary values, but they still seemed to contribute to poor performance. One can intuitively conclude why this is being that regression fundamentally fits continuous data better than discrete data which is better used over classification tasks.

While the results are not included here, I performed a series of trace runs where I dropped features from the data that were not significant contributors. Specifically, I removed features from the regressor that did not contain an absolute value greater than 0.10. This actually helped the regressor generalize better and produced a simpler model. My results gave me enough evidence in support of **Hypothesis 1**.

Table 1 - Summary of Logistic Regressor

Datasets	β_0	β_1	β_2	β_3	β_4	β_5	β_6	β_7	β_8	β_9	β_{10}	β_{11}	β_{12}	β_{13}	β_{14}	β_{15}	MSE
Car	-2.55	-0.38	-0.24	-0.03	1.41	0.25											1.53
House Votes	-0.04	-2.28	-1.17	3.76	0.07	0.49	-0.12	0.10	0.19	2.2	-1.67	-1.55	2.97	1.80	2.2	-0.70	2.85
Breast Cancer	0.02	-0.79	2.78	-0.25	-1.36	0.02	0.68	1.00	0.06	-2.64	-0.89						1.21

Linear Network

Results for the linear network are shown below. Again, the selection of explanatory variables was leveraged from heuristics discovered in Projects 2 and 3. The linear network for the *Abalone* dataset showed an accuracy of 87.15% in correctly estimating Abalone life based on the variables *Whole weight* and *Shucked weight.* For the *Machine* dataset, only 25% accuracy was achieved in correctly predicting Machine performance with the *Machine Cycle Time (MYCT)* and *Model Name* variables. This may be due to the fact that the regression slopes were nearly parallel or because the *Model Name* variable, when encoded as a numeric categorical variable, loses valuable information. For the *Forest Fires* dataset, the linear network only predicted *Burn Area* from the *FFMC* and *DMC* indices with 18.75% accuracy. The regression lines through each of the two explanatory variables showed very low correlations, which I suspect contributed to such low results. **Table 2** shows the resulting weight W_f and bias β_f of the final linear classifier. Results for regression lines through each of the two classifying features are also included to gain more context about how the classifier relates to trend of the data in general. Although my strategy could use some refining my findings give me enough evidence to not reject **Hypothesis 2**.

Table 2 - Summary of Linear Network

	Feature 0		Feature 1		Final Clasifier	
Datasets	slope_0	intercept_0	slope_1	intercept_1	W_f	β_f
Abalone	4.3371	-0.9689	3.9557	-0.2210	4.1464	39.9995
Machine	-0.1366	0.1816	0.4025	0.0371	0.1329	0.0009
Forest Fires	0.4312	0.1667	-0.0244	0.1286	0.2034	0.1941

Kordel France
Project 4
Class 605.649 Section 84
14 November 2021

Autoencoder

I constructed a 2-layer neural network for use as an auto encoder. Each of the input and hidden layers within the autoencoder were fully connected, contained ten neurons, and used the Sigmoid (logistic) activation function. For classification tasks, the Softmax activation function performed on the output layer; linear activation was performed on the same layer for regression tasks. The use of a hyperbolic tangent activation function was investigated as well, but it showed to provide erratic loss behavior on epochs greater than 30. Each layer also leveraged cross-entropy loss and mean squared error for classification and regression tasks respectively as specified by the assignment. Both Stochastic Gradient Descent and Adam were evaluated as the optimizing functions. In the end, Adam (with a learning rate of 0.002 and β=0.5) proved more effective in consistently converging over each trace run for each dataset —Adam showed better generalization. My goal with the encoder was to build it to reduce dimensionality , soI elected to have $m - 2$ neurons for the hidden layer in order to show this; $m / 2$ neurons showed some convergence in loss but did not seem to generalize well across all six datasets. I suspect that by gradually decreasing the number of neurons to $m / 2$ through the use of more layers would show great results, but the immediate jump from m neurons to $m / 2$ neurons did not yield acceptable results for my algorithm.

Table 3 shows a summary of epochs of convergence (EoC) for the final autoencoder design over each dataset. The losses have all been normalized and rescaled over a range of zero to one in order to show convergence rates over the same graph. The *forest fires* data set, for example, showed training losses on the scale of millions while the *house votes '84* dataset showed losses on the order of less than one, so normalization is necessary to allow for easier comparison on the same graph. I performed trace runs with K-fold cross validation over 1000 epochs and took the average loss across each of the five folds. I

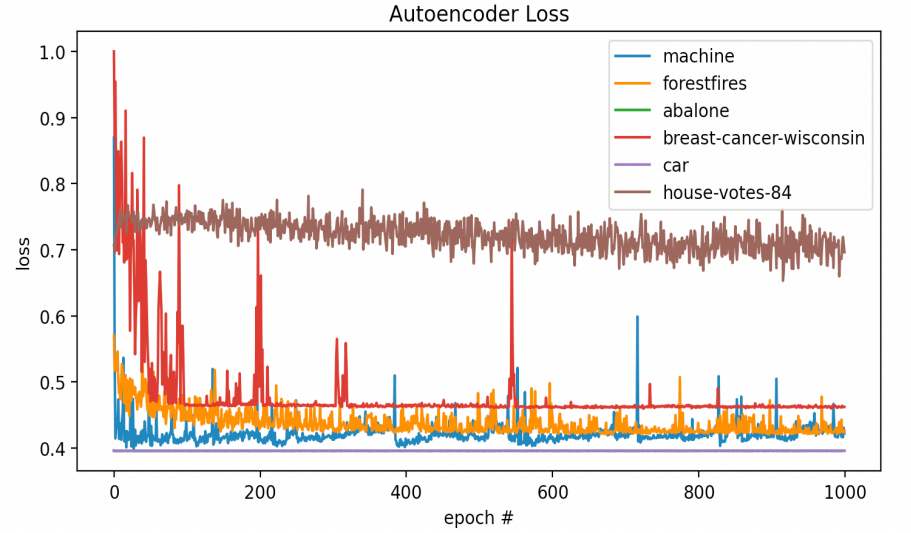

Figure 1: *Loss values for each dataset over 1000 epochs.*

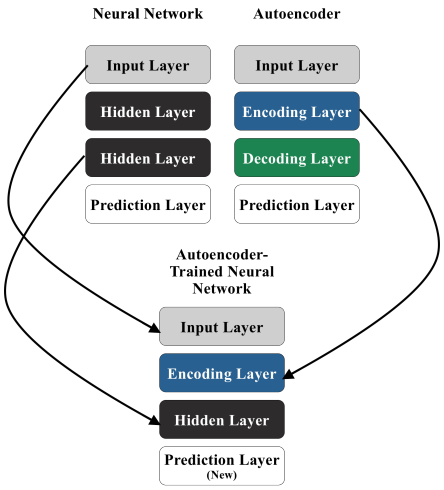

Figure 2: *Diagram illustrating the procedure performed to combine the encoder with the neural network.*

had difficulty finding an optimal number of epochs that worked for each dataset without overtraining so I programmed early stopping into the training algorithm to allow the autoencoder to stop training once it detected convergence. I defined a converging algorithm as one in which the the last ten epoch losses had not exceeded one standard deviation of the moving average of those ten losses. I have plotted training losses over the course of 1000 epochs for each dataset, but not each dataset converged at 1000 epochs. This graph is a separate trace run I performed showing each dataset over 1000 epochs to illustrate how overfitting began to occur over some datasets. It also reinforces the necessity of my early stopping algorithm.

One will notice some interesting patterns here. Both the *Abalone* and *Car* datasets converged after ten epochs, so their plots show flat on the graph because of this (the *Abalone* plot lies directly behind the *Car* plot, but the normalization of the data obscures one plot behind the other). This was surprising given that the *Abalone* dataset is one of the more complex datasets. However, what this shows is that there is a very clear correlation between the explanatory and response variables for *Abalone*. In other words, it is easier for the autoencoder to encode and decode data from this dataset because there are more obvious patterns present.

Note the sharp spikes that occur at later epochs for datasets that converged early. For example, the *Machine* dataset converged after 210 epochs but, when training continued, loss between the encoder and

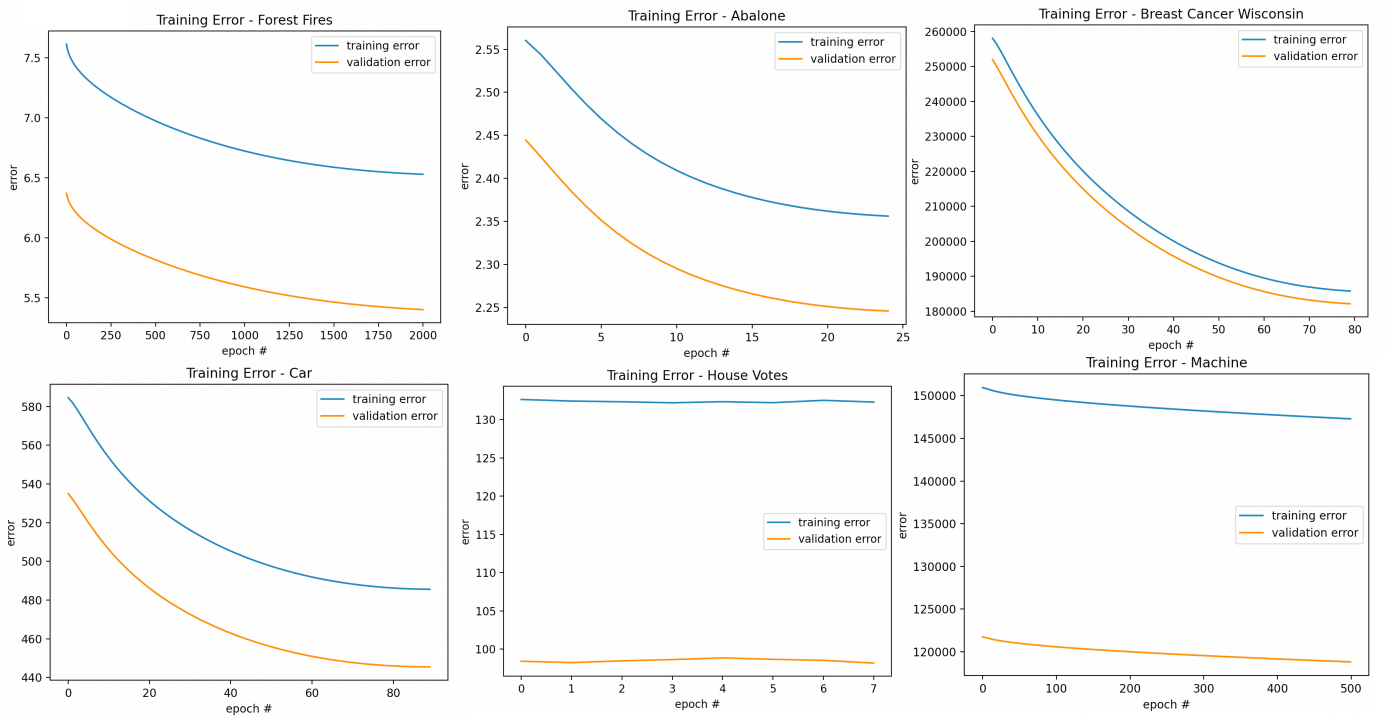

Figure 3: *Results of the non-autoencoder neural network. Note that the error represents cross entropy loss for classification datasets and mean squared error for regression datasets.*

decoder actually increased and became more volatile. The loss at 1000 epochs is actually much higher than the loss at 210 epochs—a sign of overfitting.

The *House Votes '84* dataset showed difficulty in achieving acceptable results with the autoencoder. I originally suspected a fault within my encoding procedure because the entire dataset is categorical and must be encoded. However, upon altering my encoding procedure, good results were still elusive. In ***Figure 1***, one will notice that here is a slight negative trend in the loss which indicates that training beyond 1000 epochs could perhaps yield converging results, but this seems suspicious given that the dataset is one dimensional. As previously mentioned, the encoder on all other datasets performed dimensionality reduction, so I tried increasing the number of neurons in the hidden layer for this dataset only to see if increasing dimensionality would aid in convergence. This also failed to provide convincing results. Unable to find a resolution that converged the loss faster, I suspect that these results could be due to inherent complexities within the dataset or a fault with my autoencoder.

Neural Network

A neural network with two hidden layers was trained with a number of neurons in each layer equivalent to the number of features in the input data. The reason for this is because this seemed to generalize better across all of the datasets. Increasing the number of neurons in each layer seemed to work on some datasets, but not on others. My original intent was to decrease neurons with each successive layer in

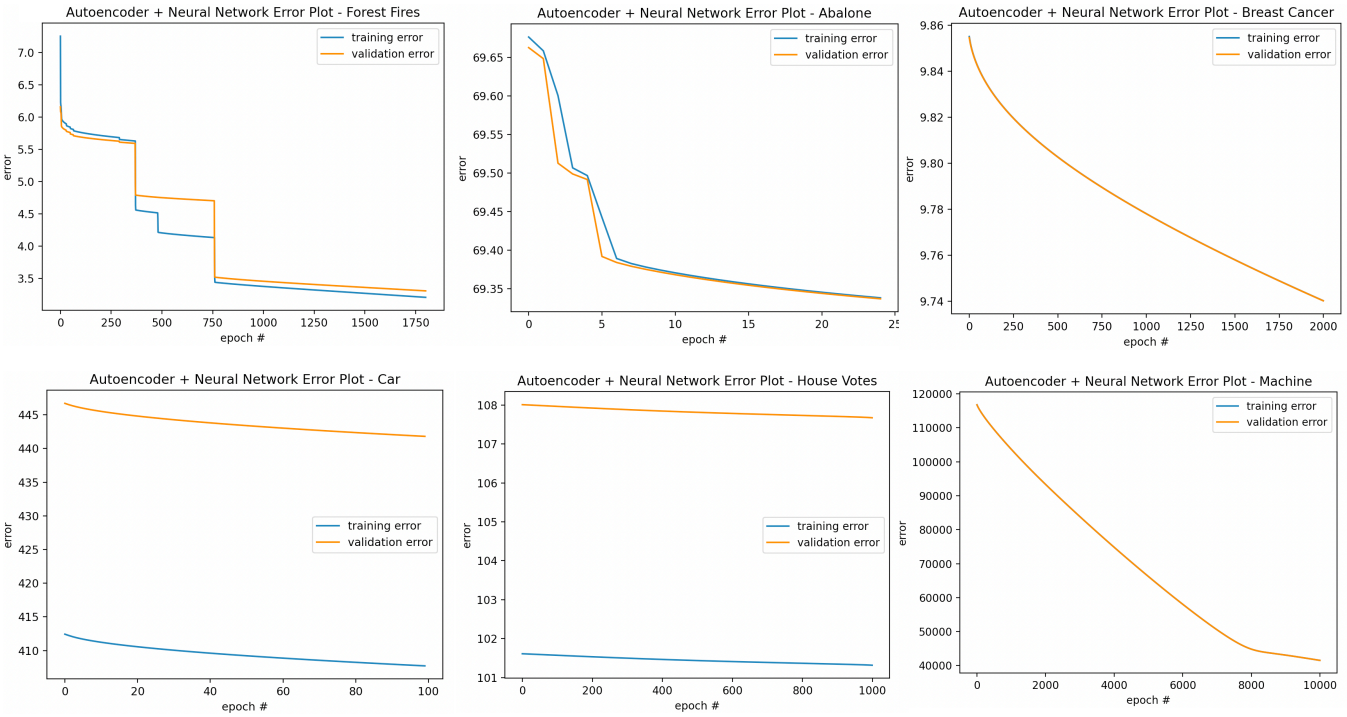

Figure 4: *Results of the autoencoder-trained neural network. Note that the error represents cross entropy loss for classification datasets and mean squared error for regression datasets. Notice the sharp change in the loss gradient for the Machine dataset around epoch 8000. We discuss this in the "Backpropagation" section.*

order to reduce dimensionality of the data, but I obtained erratic losses on some classification datasets during training. Although I am not reducing dimensionally with each layer, by keeping the number of neurons per layer constant, I am essentially gaining more detailed knowledge about the data over the same dimensionality. This can be viewed as horizontal knowledge growth instead of vertical knowledge growth. Results for training can be seen in **Figure 3.** The number of epochs needed for training the neural network varied across each dataset, just as it did with the autoencoder. While batch size remained consistent across each autoencoder, I found that varying the batch size among datasets worked best here and those values can be viewed in **Table 3.** Unless otherwise specified, the developed neural network preserved the same characteristics and hyper paremeters of the autoencoder.

Neural Network - Removing the Decoder

After the autoencoder was trained, I removed the output layer from the decoder. **Figure 2** gives a diagram illustrating this more succinctly. This was made much easier by constructing the autoencoder such that the encoder and decoder were separate modules that could be decoupled with a single function call (*autoencoder.removeDecoderOutputLayer( )*). At this point, the network has only two layers—an input layer and a hidden layer. The autoencoder provided an hourglass-like architecture in which dimensionality was decreased with each layer in the encoder, and then increased back up to input dimensionality with the decoder. I took a different approach when constructing the last part of the feedforward neural network. Instead of changing the number of neurons for each subsequent layer, I kept them the same as the input neurons similar to that of the previous neural network. The next layers added were a simple fully connected layer with Sigmoid activation and a final output layer that used either Softmax activation for classification tasks or linear (rectified linear unit) activation for regression tasks.

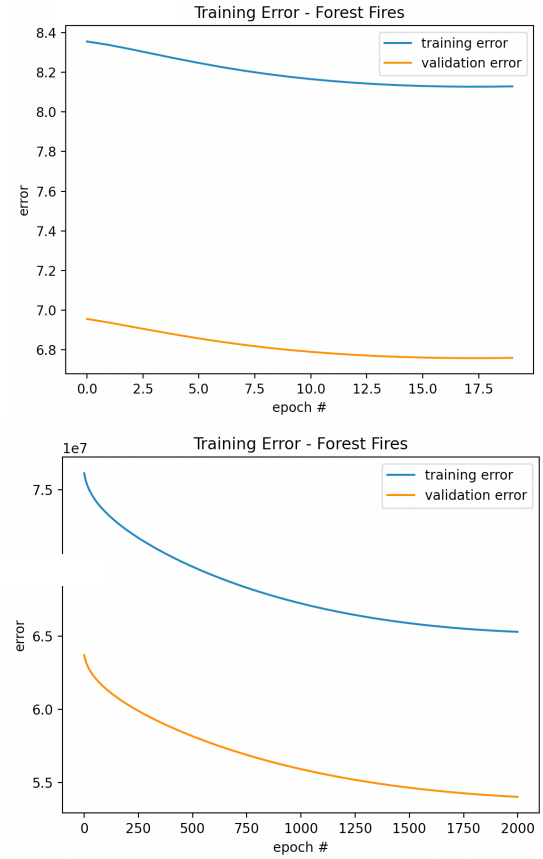

Figure 5: *Verification of performance increase with additional layers added to the network. (Left) A model trained with only one hidden layer. (Right) A model trained with two hidden layers.*

A few key points of discussion arose with training of the autoencoder-trained neural network. In comparing the results from **Figure 3** to **Figure 4,** one will notice that the distance between training and validation losses are much closer. This suggests that the layer trained by the autoencoder provides the model with better generalization over data it has not seen before. Of those models whose losses did not converge closer together, their losses are lower than their counterparts trained on the previous neural network. In light of

this, we have evidence to suggest that the encoded layer helped achieve better accuracy over the training and validation data.

These benefits provided by the encoded layer may come with a cost. Over some of the datasets (namely *Breast Cancer Wisconsin, House Votes '84,* and *Machine*), the number of epochs the model took to converge was significantly higher than the number it took to converge over the previous neural network. This allows one to suggest that, at the cost of more compute time, an autoencoder-trained neural network can provide better model performance over the same dataset and same number of layers as a regular neural network. Leveraging this argument and those of the preceding sections, I conclude that I have enough evidence to support **Hypothesis 3.**

I wanted to verify that the layers I added were actually adding value and not overfitting so I decoupled the last hidden layer from the network and analyzed the performance over the *Forest Fires* dataset. I received much faster convergence over this simpl was much higher than that received by the network with the extra This gave me confidence that added layers were preserving mor over the data without overfitting.

Backpropagation

Backpropagation proved to be much more impactful than I anticip to work effectively in the beginning, but I learned much more about how "Backprop" actually worked as I debugged my code. The principle is actually quite simple; the complexity comes in when computing the gradient and ensuring each layer is updated appropriately. To this latter point, I found it useful to perform stringent unit tests over the Backprop algorithm as I refined it.

To verify that my Backpropagation algorithm worked, refer to **Figure 6**. This figure shows the effects of the algorithm as it is omitted and included in the neural net. Very little loss adjustment occurred (so little, in fact, that it doesn't even appear to decrease when viewing the plots) when it was omitted from training. The gradient was easy to compute due to the low dimensionality of the input data, but it is easy to see how gradient computation can become difficult to track as

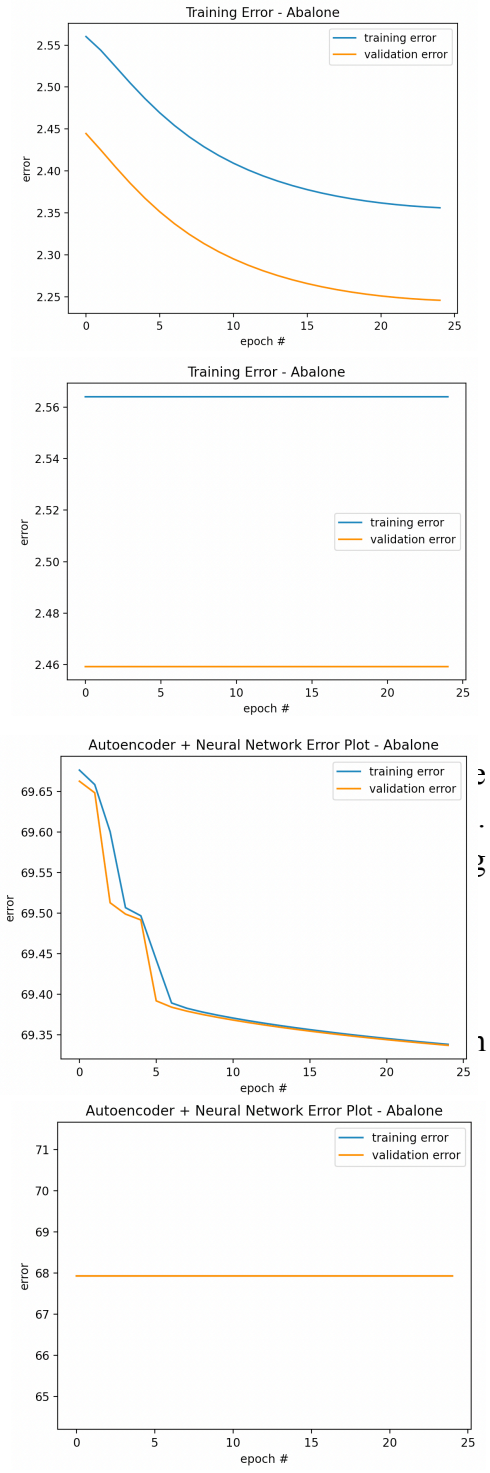

Figure 6: *Effects of disabling Backprop. (Top) Neural net with Backprop, (Top Middle) Neural net without Backprop. (Bottom Middle) Neural net + autoencoder with Backprop. (Bottom) Neural net + autoencoder without Backprop.*

dimensionality scales beyond R^2. Since I performed batch learning, weight adjustment occurred after each batch. It is difficult to verify how well Backpropagation would have worked between different batch sizes since they varied between datasets and were effectively a "tuning" parameter. I experimented with varying the batch size has a function of the epoch number to see if, by decreasing or increasing the number of time Backpropagation occurred within each epoch, I could notice a difference in the loss gradient or break through local extrema[1]. Changes of the activation function, I imagine, can also help with this.

Upon the reception of the above results for Backpropagation algorithm, I found enough evidence to support **Hypothesis 4** and not reject it.

Conclusion and Lessons Learned

Throughout the paper, I validate each of my four original hypotheses and show the efficacy behind each of the proposed algorithms. I look at isolated cases of a simple autoencoder and neural network and discuss their performance. Then, I investigate the ability of pre-trained layers from that autoencoder to provide better performance for the neural network. Additionally, I briefly explore some simpler machine learning algorithms, showing how logistic regression and a linear network can effectively work as a classifier and regressor, respectively. Throughout each section, I note any values that required tuning and how I approached the tuning process. Some results show very interesting insights about the data that weren't previously manifested with my previous *K-Nearest Neighbors* and *Decision Tree* approaches, while others show that some of the algorithms presented could use some refinement.

A key insight I gathered from this project was regarding how critical it is to ensure data dimensionality is coded correctly in order to obtain results that are not erroneous. My most time-consuming task throughout this project was debugging why data failed to transfer between layers or why Backpropagation ceased to update — all due to failure to transpose a matrix or stratify data correctly as

Table 3 - Summary of Training Results Across All Neural Network Algorithms

	Autoencoder		Neural Network		Autoencoder+Neural Network		Final Performance	
Datasets	EoC	Batch Size	EoC	Batch Size	EoC	Batch Size	MSE	Acc (%)
Abalone.csv	10	50	25	10	25	20	0.3645	
Breast-cancer-wisconsin.csv	221	50	80	20	2000	40		78.4889
Car.csv	10	50	90	30	100	30		75.2776
Forestfires.csv	433	50	2000	10	1800	20	2.8533	
House-votes-84.csv	1000	50	8	20	1000	20		43.6701
Machine.csv	210	50	500	20	2000	20	2.2172	

[1] The *Machine* dataset was a good candidate for this approach. One will notice by its loss plot in **Figure 4** during the autoencoder-based neural net training that there is a stark difference in the gradient around epoch 8000. My intent was to vary the batch size and weight of Backpropagation as epochs increased to smooth out loss curves such as these.

it was fed into the neural network. This is a key knowledge point for me as an engineer to ensure that I spend the time to ensure the data pipeline is constructed correctly before I even begin thinking about coding neural network layers and activation functions. My neural nets also pointed out holes within my encoding algorithm so, as a secondary objective, I spent some time upgrading my ordinal and nominal encoding routines to try to provide a stable processing pipeline for the network.

Constructing a neural network from scratch is something every machine learning engineer should have to perform as a rite of passage. Too many resources exist online where a neural network can be written in ten lines of code through the use of frameworks. I was able to gain a much larger appreciation for the use of these frameworks by constructing my own through this project. I also have a much more thorough understanding of the fundamental mathematics and algorithmic properties behind these advanced learning algorithms that I wasn't able to gain from the class curriculum itself.

Underneath all the code, a neural network is really an eloquent collection of simple mathematical functions implemented in a meticulously structured manner. Throughout this project, it became very sobering to see how such a sophisticated solution can be derived from fundamental computer science principles. As an engineer, I feel much more confident in being able to explain the exact operation of these "black box" algorithms due to the knowledge points gained from this project.

References

The following items were used as references for the construction of this project.

1) Cormen, T. H., & Leiserson, C. E. (2009). *Introduction to Algorithms, 3rd edition.*
2) Miller, B. N., & Ranum, D. L. (2014). *Problem solving with algorithms and data structures using Python (2nd ed.).* Decorah, IA: Brad Miller, David Ranum
3) Alpaydin, Etham. *Introduction to Machine Learning*. Fourth Edition. 2020, Massachusetts Institute of Technology.
4) Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron. *Deep Learning.* 2016, Massachusetts Institute of Technology. p 147 -149, 525 – 527.
5) Russel, Stuart J.; Norvig, Peter. *Artificial Intelligence: A Modern Approach.* Third Edition. 2015, Pearson India Education Services Pvt. Ltd. p 961-962.
6) Wikimedia Foundation. (2021, October 28). *Backpropagation*. Wikipedia. Retrieved November 14, 2021, from https://en.wikipedia.org/wiki/Backpropagation.
7) **(Code Only)** Baweja, C. (2020, March 25). *A complete guide to using progress bars in Python*. Medium. Retrieved November 14, 2021, from https://towardsdatascience.com/a-complete-guide-to-using-progress-bars-in-python-aa7f4130cda8.