Team 36 Project 3 Design Document

Max Hymer HYMERMAX1@GMAIL.COM
Department of Electrical and Computer Engineering
Montana State University
Bozeman, MT 59715, USA

Michael Downs MICHAELD20312@GMAIL.COM
Division of Computer Science
Montana State University
Bozeman, MT 59715, USA

Date: October 21, 2024

1 System Requirements

1.1 Requirement One:

Download the six (6) data sets from the UCI Machine Learning repository. You can download from the URLs above or from Brightspace. You can also find this repository at http://archive.ics.uci.edu/ml/ This requirement is very straightforward - we will simply download the breast cancer, glass, soybean (small), abalone, computer hardware and forest fire .data and .name files from Brightspace. This is an important step in analyzing how the datasets are oriented so that we can fit our pre-processing algorithm to the data.

1.2 Requirement Two:

Pre-process each data set as necessary to handle missing data and normalize as needed. We intend to read in the data and then replace all instances of "?" (as that is what marks an incomplete data point in the relevant data set (breast cancer)) in the data with a random value within the constraints of the feature (i.e. if the feature contains integers from one to ten, we replace any "?" within that feature with a random integer from 1-10). This is a simple yet effective format of data imputation that produced relatively low noise within our data. Our hope in this action is to introduce noise into the system that may or may not already be present in order to make our learning algorithm more resilient to outlier information. We intend to deal with categorical features using one-hot coding. We feel that the simplicity of this approach is a strength; we are less likely to create errors or accidentally confound our data with this strategy than we would be with a more complex one. So, we hope to spend less time troubleshooting our preprocessing procedures with this approach. Basically, we feel that this process is best because it will allow us to focus on the algorithm and hyperparameter tuning rather than prepossessing; leading our

learning function to a greater understanding of the data. This method adds a layer to the experiment as well; it allows us to compare the algorithm's performance on relatively unaltered categorical data to its performance on unaltered continuous data. We intend to normalize continuous features by applying min-max normalization using

$$x_{\text{norm}} = \frac{x - \min(X)}{\max(X) - \min(X)}$$

where x_{norm} is the normalized value, x is the original value, and $\min(X)$ and $\max(X)$ are the minimum and maximum values of the feature X. The result will scale the values of x to the range [0, 1]. We are choosing min-max normalization because it will ensure that all features contribute equally to the model, regardless of units or scales. It will also improve our convergence speed because a neural network is inherently a gradient-based algorithm.

1.3 Requirement Three:

Implement a multi-layer feedforward network with backpropagation learning capable of training a network with an arbitrary number of inputs, an arbitrary number of hidden layers, an arbitrary number of hidden nodes by layer, and an arbitrary number of outputs. In other words, the number of inputs, hidden layers, hidden units by layer, and outputs should be furnished as inputs to your program. Be able to specify whether a node uses a linear activation function for regression or a softmax activation function for classification. We plan to create a neural network that accepts the number of inputs, number of hidden layers, an array specifying the number of hidden units per layer, number of outputs, activation function type (linear or softmax for the output layer based on whether the data set is classification or regression), learning rate, whether or not to use momentum (boolean), and a momentum coefficient as parameters in the constructor. This way, the neural network will be very flexible and customizable, which will allow us to perform our experiments in a very methodical and thorough manner. It will allow for very complex tuning as well. The neural network will first intialize weights randomly (within a weight matrix) for each layer to ensure that each neuron computes a unique output and contributes differently to the learning process. We will use uniform initialization (between 0 and 1) to keep this step simple. Each weight matrix will contain the connection weights between each node in the current layer and each node in the next layer. If there are n input features, h_1 nodes in the first hidden layer, h_2 nodes in the second hidden layer, h_f nodes in the final hidden layer, and o output features, the weight matrix between the input layer and the first hidden layer will have $n \times h_1$ entries, the weight matrix between the first hidden layer and the second hidden layer will have $h_1 \times h_2$ entries, and the weight matrix between the final hidden layer and the output layer will have $h_f \times o$ entries. Each weight in these matrices represents the strengths of the connection between a node in the first layer and a node in the second layer (e.g. each weight in the matrix between the input layer and the first hidden layer represents the strength of

the connection between an input feature and a node in the first hidden layer). Then, the neural network will begin the training loop, starting with a forward pass over the data. To start the forward pass, each neuron in a layer computes the weighted sum of its inputs. This weighted sum, denoted as z, is calculated as $z = W \cdot x + b$ where W is the weight matrix for the layer, x is the input vector (either from the previous layer or the original input to the network), and b is the bias vector for the neuron in the layer. So, $W \cdot x$, a weighted sum of inputs for a neuron, plus b, the bias term, produces the net input to each neuron in the layer. Then, the sigmoid function (a.k.a. the logistic activation function) is applied using

$$a = \sigma(z) = \frac{1}{1 + e^{-z}}$$

where a is the output after applying the activation function and $\sigma(z)$ is the sigmoid function, which squashes the weighted sum z into a value between 0 and 1. This introduces non-linearity into the model, which allows the neural network to model complex, non-linear relationships in the data. This is the crux of the algorithm; it would simply be a series of linear transformations without it. This process continues through all of the hidden layers until the output layer is reached. For this output layer, a different activation function is chosen based on whether model's data set is a classification set or a regression set. For classification sets, we will use the softmax activation function

$$a_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

where a_j is the output for the j^{th} class, z_j is the weighted sum for that class, and K is the number of classes. For regression sets, we will use a linear activation. Since our weighted sum z provides a linear output, it becomes our linear activation. So, regression sets essentially just won't use an activation function in the output layer. After this forward pass, our neural network will use a backpropagation algorithm to adjust our weights based on loss between the predicted output and the actual target value. First, this algorithm will compute the loss for the network's predictions. For regression, it will use Mean Squared Error (MSE) Loss

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

where y_i is the actual value, $\hat{y}_i$ is the predicted output, and n is the number of samples. For classification, it will use Cross-Entropy Loss

$$\text{Cross-Entropy Loss} = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

where y_i is the actual label and $\hat{y}_i$ is the predicted probability for class i. Then, it will apply the calculus chain rule to calculate the gradient of the loss function with

respect to each weight in the network and propagate the error backward through each layer in the network. First, the gradient of the loss with respect to the output layer's weights is calculated with

$$\frac{\partial \text{Loss}}{\partial \hat{y}_i} = \hat{y}_i - y_i$$

which gives the error for each output neuron. This is used to compute the gradient with respect to the weights connecting the final hidden layer to the output layer. Next, this error is backpropagated through the hidden layers. For each hidden layer, we will compute the gradient of the loss with respect to the weights connecting the current layer to the next layer, which looks like

$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z) \cdot (1 - \sigma(z))$$

for the sigmoid activation function (which is the activation function we plan on using within our hidden layers). Then, we will apply the chain rule to compute the gradients for the rest of the layers. Lastly, once the gradients have been calculated, the neural network's weights are updated in order to make it more accurate. We will do this with the Gradient Descent method. In this method, the weights are updated by subtracting a fraction of the gradient (controlled by the learning rate) from the current weight values

$$W_{\text{new}} = W_{\text{old}} - \eta \frac{\partial \text{Loss}}{\partial W}$$

where η is the learning rate and $\frac{\partial \text{Loss}}{\partial W}$ is the gradient of the loss with respect to the weights. The weights are updated starting with the output layer and moving backwards towards the input layer, layer by layer. We plan to include an option to update the neural network's weights with momentum as well. This option, if turned on, will speed up convergence and reduce oscillations in the learning process because it will change the weight update calculation to include a fraction of the previous weight update. With this option, the update rule is

$$\Delta W_t = -\eta \frac{\partial \text{Loss}}{\partial W} + \alpha \Delta W_{t-1}$$

where α is the momentum coefficient (a value between 0 and 1) and ΔW_{t-1} is the previous weight update from the last iteration. Therefore, the new weights are updated using

$$W_{\text{new}} = W_{\text{old}} + \Delta W_t$$

when momentum is on. After this final step in the training loop is completed, the loop will repeat until the loss converges (reaches a point where it no longer decreases significantly between epochs). So, we will not be defining a fixed number of epochs or loss threshold; we will run our algorithm as many epochs as it takes for the loss to converge. We believe that this will be the most thorough and accurate way to train our neural network because it allows the algorithm to run until ultimate completion.

1.4 Requirement Four:

The hidden nodes should use a sigmoid activation function (you may choose between logistic or hyperbolic tangent). Do not use ReLU or other similar activation functions. Remember that these choices affect the update rules because of having different derivatives. Implement learning such that momentum is provided as an option. As described in Requirement Three, we plan to use the logistic activation function to introduce non-linearity into our hidden nodes. We understand that this choice affects our update rule because the gradient of the loss with respect to the weights connecting the current layer to the next layer is calculated with the partial derivative of our chosen activation function. We plan to implement learning such that momentum is provide as an option as described in Requirement Three.

1.5 Requirement Five:

Develop a hypothesis focusing on convergence rate and final performance of each of the different network architectures for each of the various problems. We plan to develop hypotheses based on our knowledge of each data set along with our knowledge of how each network architecture functions. With this information, we feel we can predict with at least a reasonable degree of accuracy the final performance and convergence rate of each of the chosen network architectures on each of the data sets.

1.6 Requirement Six:

Test the MLP (backpropagation) algorithm on networks with 0, 1, and 2 hidden layers. Your experimental design should apply 10-fold cross validation. We plan to tune all of our hyperparameters before we test our backpropagation algorithm. We will perform the following process on all six data sets assigned for this project. First, we will tune the learning rate and momentum term for a network with 0 hidden layers. Then, we will test our backpropagation algorithm with 0 hidden layers with the learning rate and momentum term that performed best in our tuning process. After that, we will tune the learning rate, momentum term, and number of hidden nodes per layer for a network with 1 hidden layer. We will use the rule of thumb that we should have fewer hidden nodes than inputs to pick our tuning values for this hyperparameter. Again, we will use our hyperparameters that performed best to test our backpropagation algorithm, this time with 1 hidden layer. We will repeat this process to tune and test our backpropagation algorithm on a network with 2 hidden layers, but this time we will use the rule that we should need less hidden nodes per layer than with 1 hidden layer to pick our tuning values for the corresponding hyperparameter. We will use loss functions to evaluate the performance of our tests. Specifically, we will use mean squared error to evaluate regression data sets and cross entropy loss to evaluate categorical data sets. We will keep track of the number of epochs to convergence as well in order to find correlations between performance and convergence rate. We will use the 10-fold cross validation strategy described in the

Project 2 overview to perform our experiment. For tuning, this strategy involves taking out a stratified 10% of the data and using it as the test fold, splitting the remaining data into 10 stratified folds, removing 1 fold, and combining the remaining 9 folds into training data until each fold has had a turn being the removed fold. For testing, this strategy involves splitting the data into 10 stratified folds, removing 1 fold to use as the test fold, and combining the remaining 9 folds into training data until each fold has had a turn being the test fold.

1.7 Requirement Seven:

Write a final paper encompassing the results of our tests. This will be an extremely similar document to this one, with all of our testing, reflection on the hypothesis, and analysis of our data structuring, flow, and test strategy.

1.8 Requirement Eight:

Make a video demonstrating the results of our code. This video will focus on the behavior of our code, will show input, data structure, and output, will be less than five minutes in length, will be posted on Microsoft Stream, will only employ fast-forwarding through long computational cycles, and will contain verbal commentary explaining the elements we are demonstrating. As far as project three specifics, our video will: show sample outputs from one test fold showing performance on one classification network and one regression network, including results from zero, one, and two hidden layers, show a sample model for the smallest of each of our neural network types, including weight matrices with the inputs/outputs of the layer labeled accordingly, show and explain how an example is propagated through a two hidden layer network, including the activations at each layer being calculated correctly, show the gradient calculation at the output for one classification network and one regression network, show the weight updates occurring on a two-layer network for each of the layers for one classification network and one regression network, and show the average performance over the ten folds for one of the data sets for each of the types of networks.

2 System Architecture

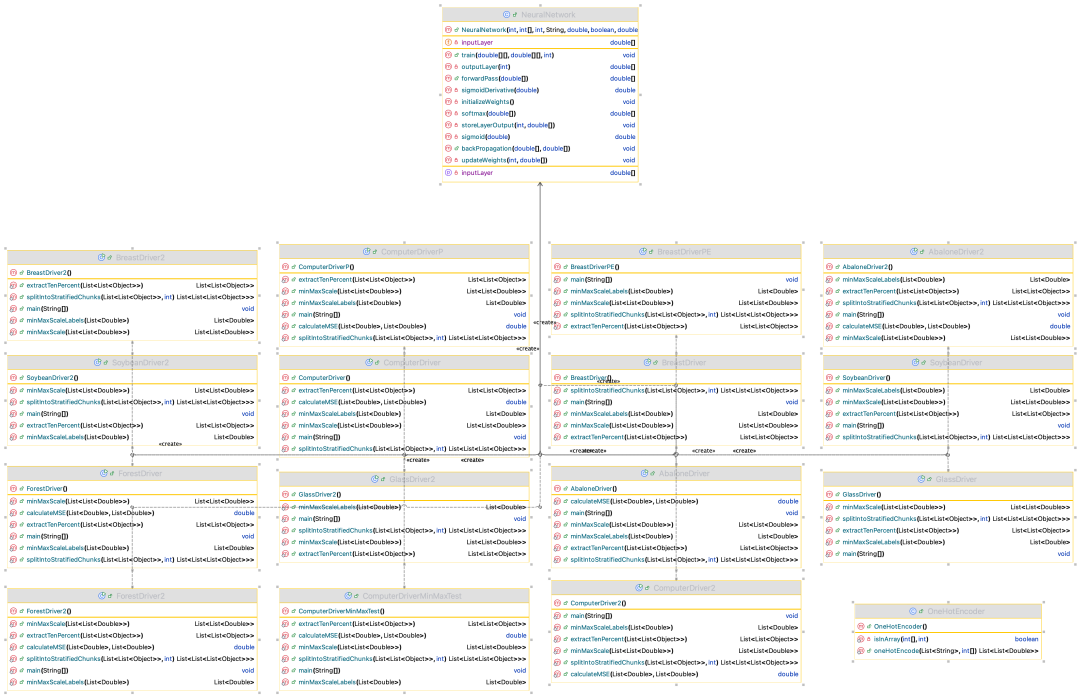

Figure 1: Project 3 UML Diagram

2.1 Drivers

We created two unique drivers for each data set that read in the corresponding data set, perform unique preprocessing based on the needs of that data set (which includes calling the one-hot encode method (which is its own external class) or the min-max scale method (which is implemented internally within each driver) if needed), perform 10-fold stratified cross validation on the data set (the splitIntoChunks method is used to split the data set into 10 folds), call the Neural Network algorithm for each fold, calculate our cross entropy and mean squared error for each fold, and calculate and print the average across all folds for these loss functions. The difference in the two drivers is in the way 10-fold cross validation is performed; one driver performs tuning 10-fold cross validation and the other performs testing 10-fold cross validation (both described in Requirement Six). This is done to simplify data gathering for the results section of the paper will we write describing our findings. There are also drivers that will be used for the video (they contain many more print functions) within the UML diagram, which is why it contains 15 drivers rather than 12.

2.2 One-Hot Encoder

Our one-hot encoder class is designed to one-hot encode any categorical data present in the required data sets. It takes the initially preprocessed data and and an array containing the indices of the categorical columns within the array as arguments (so it knows which columns to one-hot encode) and returns a list containing both the newly one-hot encoded data and the already continuous data.

2.3 Neural Network

Our neural network design is outlined in Requirement Three.

3 System Flow

In our drivers, we will be taking all of our datasets in our program and creating representations of them depending on how they may need to be changed to all be processed by the same algorithm. For low occurrences in unknowns, ignoring them, or removing them is recommended. For our project, we plan on taking some random information within the bounds of the feature and implementing it at the unknown position. If present in preprocessing, we will be one hot encoding categorical data so that it can be accounted for within our learning algorithm. We will normalize our data by applying min-max normalization (if appropriate) at this step as well. Then, we will be using stratified 10-fold cross-validation to test our data in a manner that represents the dataset as a whole within each fold. We will tune and test using the 10-fold cross validation strategy outlined in both the Project 2 overview and towards the end of Requirement Six. For each fold, we will train the neural network with the training set (the 9 combined folds) and test it with the testing set (either the 10% or hold-out fold). The flow of the training loop within the neural network is outlined in Requirement 3. After all of our training and testing is finished, information regarding calculations of our loss functions and calculations of outputs will be displayed. After evaluating our loss functions, reflection and problem solving will take place, but this is the general flow of the data through our system.

4 Test Strategy

Our general testing strategy is outlined in Requirement Six.

5 Task Assignments/Schedule

$\longrightarrow$ Max Hymer
- Driver implementation (requirements 1 and 2) - 10/22
- One-hot encoding implementation (requirement 2) - 10/25
- Develop hypotheses (requirement 5) - 10/25

⟶ Michael Downs
- Min-max normalization implementation (requirement 2) - 10/25
- Neural network implementation (requirements 3 and 4) - 10/25
- Edit code to show correct printouts, etc. for the video (requirement 8) - 10/27

⟶ Working together
- Testing and tuning (requirement 6) - 11/1
- Commenting and recording video presentation (requirement 8) - 11/3
- Final paper (requirement 7) - 11/3