Team 36 Project 4 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: November 15, 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 the data to ensure you are working with complete examples (i.e., no missing attribute values). 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. 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 outputs using one-hot encoding. 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 functions to a greater understanding of the 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. This strategy will also fit the data to our constraints of our model; since our algorithms use logarithmic activation, min-max normalization was the obvious choice.

1.3 Requirement Three:

Implement feedforward neural network training using a genetic algorithm for training the weights. Use your GA to train the 0, 1, and 2-hidden layer networks you produced from Project 3. 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), and some algorithm-specific hyperparameters discussed later 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 initialize weights using Xavier Initialization (within a weight matrix)

$$W \sim \mathcal{N}\left(0, \left(\sqrt{\frac{1}{n_{in} + n_{out}}}\right)^2\right)$$

where n_{in} is the number of inputs to the layer and n_{out} is the number of outputs for the layer for each layer to ensure that each neuron computes a unique output and contributes differently to the learning process. We chose Xavier initialization because it works well with sigmoid activation functions, like the logarithmic function we chose. 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). We also plan to initialize biases with Small Random Initialization

$$b \sim \mathcal{N}(0, 0.01^2)$$

so that our model can fit data effectively; these biases will shift the activation function up or down which will allow the model to learn relationships that don't pass through the origin. 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 genetic algorithm with real-valued chromosomes to adjust our weights based on loss between the predicted output and the actual target value. But, unlike backpropagation, it won't adjust the weights within a single network. Instead, it will evaluate the fitness of the given network and will use a selection process to select the most fit networks to survive. First, this algorithm will compute the loss for the network's predictions. For regression, it will use negative 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 negative 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. We are using negative MSE and Cross-Entropy Loss because networks with higher fitness scores are considered fitter and therefore more likely to be selected for reproduction in the next generation. So, since we want networks with low MSE or Cross-Entropy Loss, we use the negative of these loss functions. Then, it will select which networks make it to the next generation. We plan to use Roulette Wheel Selection to achieve this. This method first assigns each network a probability of selection proportional to its fitness. So, networks with higher fitness scores are more likely to be selected. Then, figuratively, it spins a roulette wheel where each segment's size is proportional to a network's fitness score and determines the parent networks from a spin of this wheel. We plan to use this method as it ensures diversity in the population as less fit networks still have a chance of being selected. The next step is crossover, which is a process inspired by biological reproduction. It combines the genetic material of two parent chromosomes to produce one or more offspring with the goal of combining weights from both parents to produce a better-performing offspring. We plan to use uniform crossover, which first selects each weight randomly from either parent with a fixed probability of 50%. This method is simple, yet still highly effective, at introducing variation into the population. So, it is used to explore new regions of the solution space that neither of the parents could reach individually. Next, mutation, which is a process that introduces random changes to some of the weights in the offspring networks, occurs. This process is designed to prevent the algorithm from getting stuck in suboptimal local minima by maintaining genetic diversity within the population. It involves adding a small random value (drawn from a Gaussian distribution with a mean of zero and a small standard deviation) to a selected weight. The mutation rate, which is each weight's small probability of being selected for mutation, determines how many weights will be updated with this method. This rate is a value we will need to tune. This again expands the solution space beyond the scope visible with simple crossover. Lastly, the process of elitism is applied. This ensures that the best solutions from the current generation are preserved in the next generation with no changes, which improves the quality of the population. It works by selecting a fixed percentage of the best-performing chromosomes (neural networks with the highest fitness scores). This percentage will need to be tuned as well. Then, it puts these chromosomes directly through to the next generation, bypassing crossover and mutation. This way, good solutions are still kept while new solutions are explored through crossover and mutation. After this final step in the training loop is completed, the loop will repeat until the fitness scores converge (reach a point where they no longer change significantly between generations). So, we will not be defining a fixed

number of generations; we will run our algorithm as many generations as it takes for the fitness scores 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. This algorithm should be much better at finding optimal solutions than backpropagation; it explores far more solutions and therefore is less likely to get stuck in local minima. It also allows for parallel processing, which can speed up training.

1.4 Requirement Four:

Implement feedforward neural network training using a differential evolution procedure for training the weights. Use DE to train the 0, 1, and 2-hidden layer networks you produced from Project 3. This algorithm works the same way as the GA algorithm (aside from initialization, which will use some algorithm-specific hyperparameters along with the basic neural network parameters) until the forward pass is complete. Then, it begins its initialization process. This involves creating an initial population of potential solutions. Each solution, called an individual, consists of a full set of neural network parameters, including the weights and biases that will be used in training. These parameters are initialized within a set range, which will need to be tuned. We will perform this tuning using multiples of the number of parameters we are optimizing. This form of initialization is designed to create a large variety of solutions that cover a large region of the solution space. Once this step is complete, mutation begins. This involves creating a new solution, called a "mutant vector," for each individual in the population. To generate this mutant vector v_i for the i_{th} individual x_i, three distinct solutions (that must differ from x_i) x_a, x_b, and x_c are randomly selected from the population. Then, the mutant vector v_i is calculated using the formula

$$v_i = x_a + F \times (x_b - x_c)$$

where F is a scaling factor, usually between 0 and 1, which controls the magnitude of the difference between x_b and x_c. This factor introduces variation in the population which, again, allows the algorithm to thoroughly explore the search space. The difference $(x_b - x_c)$ produces a directional vector that highlights how one individual differs from another, and F scales this difference. Adding this to x_a creates a new, mutated solution v_i. Again, as in the GA algorithm, this process is designed to help the algorithm escape suboptimal local minima. Next, the crossover step is used to combine the mutant vector v_i with the current individual x_i to produce a new offspring u_i, ensuring that the offspring inherit traits from both the current individual and the mutant vector. For each component j of the offspring u_i, a uniformly distributed random number $\text{rand}_j(0, 1)$ is generated. If this number is less than or equal to the crossover probability CR, the component from the mutant vector v_i is chosen. Otherwise, the component from x_i is retained. Additionally, a randomly chosen index j_{rand} ensures that at least one component from v_i is included in u_i, even if

$\text{rand}_j(0,1) > \text{CR}$ for all components. The crossover probability will need to be tuned as well. Then, the selection step, which evaluates the fitness of the offspring, occurs. Just like in the GA algorithm, the negative MSE will be used to evaluate fitness for regression data sets while the negative Cross-Entropy Loss will be used for classification data sets. If the offspring u_i has a higher fitness score than the the original individual x_i, u_i replaces x_i in the next generation. This way, the population will improve in quality over time as the most fit individuals are kept and the less fit ones are discarded. After this final step in the training loop is completed (again just like in the GA algorithm), the loop will repeat until the fitness scores converge (reach a point where they no longer change significantly between generations). So, we will not be defining a fixed number of generations; we will run our algorithm as many generations as it takes for the fitness scores 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. And, for the same reasons as the GA algorithm (expanded solution space and parallel processing capability), the DE algorithm should work better than a gradient-based optimization method like backpropagation.

1.5 Requirement Five:

Implement feedforward neural network training using a particle swarm optimization algorithm to train the weights. Use PSO to train the 0, 1, and 2-hidden layer networks you produced from Project 3. This algorithm works the same way as the GA and DE algorithms (aside from initialization, which will use some algorithm-specific hyperparameters along with the basic neural network parameters) until the forward pass is complete. Then, the Particle Swarm Optimization (PSO) algorithm is used to optimize the neural network's weights. This algorithm, inspired by the social behavior of a flock of birds, employs a population of particles that "fly" through the search space to find the optimal weights for the network. Each particle represents a potential solution, which is set of weights in this case. The "position" of each particle defines the current set of weights while the "velocity" of each particle determines the direction and speed at which a particle searches for a better solution. Just like with the GA and DE algorithms, this algorithm start with an initialization process. For PSO, this process involves randomly initializing each particle's position (within the search space) and velocity (with the same dimensionality as the position). Then, similarly the other algorithms, the fitness of each particle is assessed with either negative MSE (for regression data sets) or negative Cross-Entropy Loss (for classification data sets). Each particle tracks a personal best, p_i, which is the best position (highest fitness) the particle has visited based on its own experience. It also tracks a global best, g, which is the best position (highest fitness) found by any particle in the swarm. Since this value is the most successful observed solution, it is used as a guide for the rest of the swarm. Velocity and position updates, which are based on these metrics, happen

next. The equation for these updates is

$$v_i = wv_i + c_1 r_1(p_i - x_i) + c_2 r_2(g - x_i)$$

where v_i is the velocity of particle i, w is the inertia weight, which controls the influence of the previous velocity, c_1 and c_2 are the cognitive and social acceleration coefficients, respectively, r_1 and r_2 are random values between 0 and 1, introducing stochasticity to prevent premature convergence and ensure diverse exploration, p_i is the particle's personal best position, g is the global best position, and x_i is the current position of the particle. Within this equation, a higher w encourages exploration, while a lower w promotes exploitation (focusing on promising areas) and c_1 and c_2 determine how strongly a particle is pulled towards its personal best and the global best. The inertia weight and the cognitive and social acceleration coefficients will need to be tuned here. To prevent particles from moving too quickly, velocity clamping will be applied with

$$v_i \leftarrow \begin{cases} v_i & |v_i| < v_{\max} \\ v_{\max} & |v_i| \geq v_{\max} \end{cases}$$

where $v_{\max}$ is the predefined maximum velocity. This ensures that each component of v_i stays below v_{max}. This way, the velocities are not able to grow to such a high level that they no longer have any real meaning. The value of v_{max} will need to be tuned here. The position of each particle is then updated using

$$x_i = x_i + v_i$$

in order to shift the particle to a (hopefully) more optimal position in the solution space. Unlike with the previous two algorithms, we will not be running our particle swarm optimization algorithm until convergence. Instead, we will run it for a fixed number of generations. We feel that attempting to train this model to convergence requires too much unnecessary complexity without enough upside, so we feel that using a fixed number of generations is the best solution. So, after this final step in the training loop is completed, the loop will repeat until the fixed number of generations is reached. And, for the same reasons as the GA and DE algorithms (expanded solution space and parallel processing capability), the PSO algorithm should work better than a gradient-based optimization method like backpropagation.

1.6 Requirement Six:

Develop a hypothesis focusing on convergence rate and final performance of each of the chosen algorithms. 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 (if applicable) of each of the chosen network architectures on each of the data sets.

1.7 Requirement Seven:

Compare the results of all three of your population-based algorithms and backpropagation (which should have been implemented previously), after tuning. You may reuse networks trained during Project 3 on the data sets given above as long as the folds for cross-validation are the same. Your experimental design should apply 10-fold cross validation. We plan on using stratified 10 fold cross validation and then calling all four of our algorithms from our drivers so that we can return all of our loss functions for each fold and all ten folds averaged. This should be much more efficient in recording our video and creating information that is easily comparable. We do plan to reuse our neural network from Project 3 in this experiment.

1.8 Requirement Eight:

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 our hypotheses, and analysis of our algorithm performance relative to the convergence rate (if applicable) and the different data sets.

1.9 Requirement Nine:

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 four specifics, our video will: provide sample outputs from one test fold showing performance on one classification and one regression network, including results for the two hidden layer cases only but for each of the learning methods, demonstrate each of the main operations for the GA: selection, crossover, and mutation, demonstrate each of the main operations for the DE: crossover and mutation, demonstrate each of the main operations for the PSO: pbest calculation, gbest calculation, velocity update, and position update, and show the average performance over the ten folds for one of the classification data sets and one of the regression data sets for each of the networks trained with each of the algorithms.

2 System Architecture

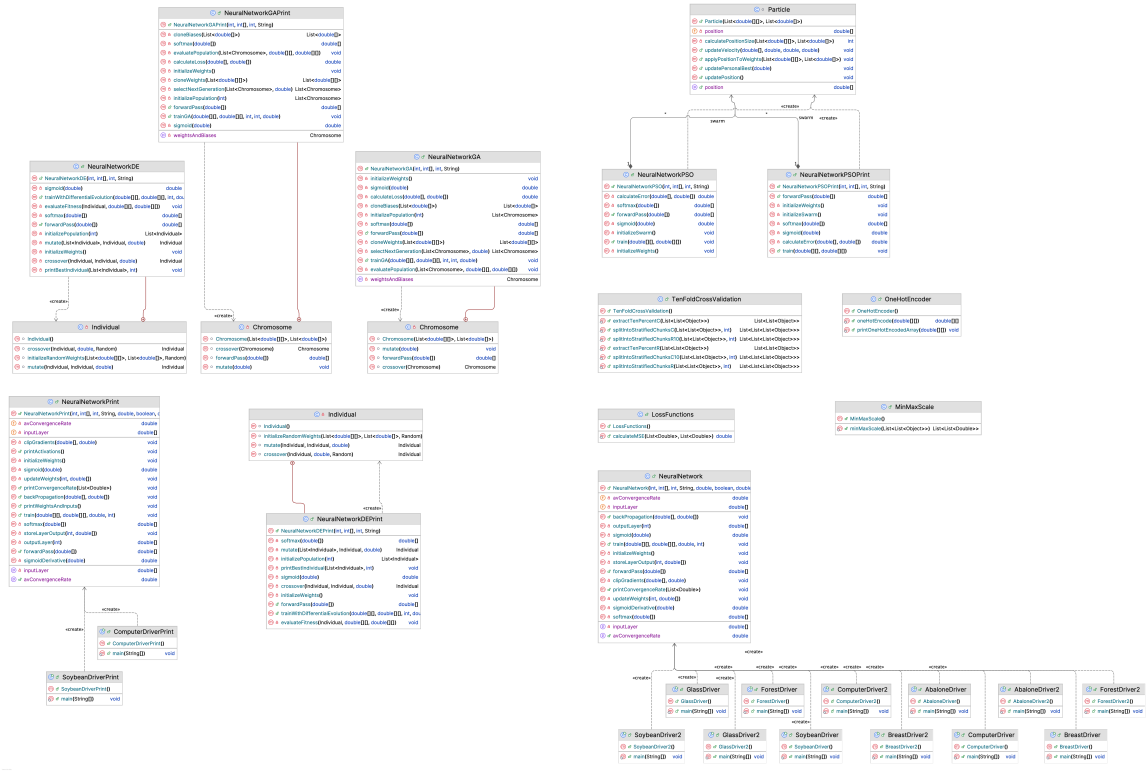

Figure 1: Project 4 UML Diagram

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, and call methods to perform the rest of our experiments. For tuning drivers, a method to extract 10% of the stratified data is called. Then, the remaining data is split into 10 stratified chunks. For testing drivers, the data is just split into 10 stratified chunks. The training and testing data sets for a fold are created and Min-Max Normalized. For classification data set drivers, the training data set outputs (labels) are one-hot encoded. Then, a neural network (Backpropagation, GA, DE, or PSO depending on the experiment) is initialized and trained with the Min-Max Normalized training data. One final forward pass of the neural network is performed with the Min-Max Normalized test data. The 0/1 Loss or Mean Squared Error for the fold is calculated from the predictions this forward pass returns. For classification data set drivers, the returned raw percentages are decoded into a predicted class before 0/1 loss is calculated. The test instance, prediction, actual value, 0/1 Loss or MSE, and Average Convergence Rate (for all neural network types except PSO) is printed for the fold.

Once the driver iterates through all of the folds, the average MSE or 0/1 Loss and Average Convergence Rate (again for all neural network types except PSO) across all of the folds is calculated and printed. The aforementioned Min-Max Normalization, Stratified 10-Fold Cross Validation (including the stratified 10% extraction), One-Hot Encoding, and MSE calculation happen within separate helper classes. The Average Convergence Rate calculation happens with each neural network (if applicable). Our print drivers are simply drivers that call the neural network print methods and include more print statements. These drivers and neural networks are utilized for debugging and recording our video, which is discussed in more detail in a future section.

2.2 One-Hot Encoder

Our one-hot encoder class is designed to one-hot encode categorical outputs present in the required data sets. It takes the initially preprocessed outputs as an argument and returns an array containing the newly one-hot encoded outputs.

2.3 Min-Max Normalization

Our min-max normalization class is designed 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].

2.4 Loss Functions

The Loss Functions class calculates the Mean Squared Error between the predicted and actual values within regression test folds. Mean Squared Error is a loss function used in regression tasks. It measures the average of the squares of the differences between the predicted values and the actual values. It is defined as

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

where y_i is the actual value, $\hat{y}_i$ is the predicted value, and n is the number of data points. MSE penalizes larger errors more heavily than smaller ones due to squaring the error value. This makes the loss function useful in measuring prediction inaccuracies. For classification tasks, the error between predicted and actual classes will be calculated with 0/1 loss. This will happen within the classification drivers rather than the Loss Functions class. This algorithm takes all of the guesses and records them as a 0 when they are incorrect and a 1 when they are correct. Then once all of

the predictions are made the total incorrect guesses are divided by the total number of guesses. This gives us a percentage of incorrect guesses that will quickly tell us how accurate the algorithm is in predicting classes based on the feature information and more precisely the percentage of estimating the class incorrectly. The formula for this is

$$L(G) = \frac{\sum I}{T} * 100$$

where L(G) is our 0/1 loss as a percentage, G is our guess, T is the total number of guesses, and $\sum I$ is our total number of incorrect guesses.

2.5 Ten-Fold Cross Validation

Our Ten-Fold Cross Validation class will perform 10-Fold Stratified Cross-Validation on our data sets. 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.

2.6 Neural Network Backpropagation

The design of our neural network using backpropagation was outlined thoroughly in requirement three of the Project 3 Design Document. Since we are using the exact same algorithm in this project (and its description is lengthy), we will not outline it again here. Please refer to the Project 3 Design Document for this information.

2.7 Neural Network Genetic Algorithm

The design of our neural network using a genetic algorithm with real-valued chromosomes is outlined in requirement three.

2.8 Neural Network Differential Evolution

The design of our neural network using differential evolution is outlined in requirement four.

2.9 Neural Network Particle Swarm Optimization

The design of our neural network using particle swarm optimization is outlined in requirement five.

2.10 Neural Network Print Classes

Our neural network print classes are just copies of our neural network classes described above. However, they include many print statements that are omitted from the above neural network classes. Essentially, they exist as debuggers for our above neural network classes and as tools to record the video described in requirement nine. We've found that creating separate classes for this video with the required print statements simplifies its creation, hence the existence of these classes.

3 System Flow

The flow of our system is, as one may expect, dictated by our drivers. Our drivers call all of the necessary functions and print all of the necessary information to run a data set through a given algorithm from start to finish. So, our system flow is outlined within the drivers section of our system architecture. We feel it would be redundant to rephrase it slightly and put it in full here.

4 Test Strategy

Our test strategy will be a mostly uniform process across the six datasets. Each dataset will be modeled with zero, one, and two hidden layers with no layer larger than the input or previous layer. First, we will perform data imputation. This is only necessary on the breast cancer identification data set. To do this, we will replace instances of "?" in the data with a random integer with a value of 1-10 as that is the range of the data and we want to keep our data imputation approach simple. Then, we will perform 10-Fold Stratified Cross-Validation. 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. We will Min-Max Normalize both the training and testing folds to make our data compatible with our neural networks. Next, we will initialize our backpropagation neural network with the appropriate input size, hidden layer sizes, output size, activation type, learning rate, whether or not to use momentum, and a momentum coefficient. We will initialize our other neural networks with basic neural network parameters and algorithm-specific parameters described in more detail in the respective algorithm descriptions as well. We then train each of our folds of information from our data sets with our backpropagation neural network, our genetic algorithm neural network, our differential evolution neural network, and our particle swarm optimization neural network, returning and printing results that are comparable. This is where reflection on our results will happen. We will be evaluating the effectiveness of our population size, crossover rate (GA),

mutation rate (GA), scaling factor (DE), binomial crossover probability (DE), inertia (PSO), cognitive update rate (PSO), social update rate (PSO), and maximum velocity (PSO), tuning and testing these hyperparameters to find the most accurate results for each learning model. We will use a wide range of hyperparameter values and networks with 0, 1, and 2 hidden layers to do this. We will also evaluate which data sets performed best with each algorithm and how the convergence rate (for all algorithms except PSO) is related to the performance of each algorithm.

5 Task Assignments/Schedule

$\longrightarrow$ Max Hymer
- Genetic Algorithm implementation - 11/22
- Abstract, introduction, creating tables - 11/25
- hypothesis reflection - 12/3

$\longrightarrow$ Michael Downs
- Driver Implementation (pass data to algorithms) - 11/17
- Swarm implementation, differential evolution - 11/24
- Testing and algorithm explanation (paper) - 12/1

$\longrightarrow$ Working together
- Testing and tuning - 12/1
- Commenting and recording video presentation - 12/4
- Recording Final testing values - 12/3
- Discussion and conclusion - 12/5

Key: Check in throughout project on timeline completion. If things are not getting done, adjust timeline and come up with a new plan.