# Project 3

Team 14
Josh Aney - josh.aney@icloud.com
Owen Cool - ohcool57@gmail.com
Nicolas Perl - nicolas.leon.perl@gmail.com
21 October 2024

---

**Requirements**

For this project, it is required to download six datasets from the UCI Machine learning repository. Once the data sets are downloaded it will be required to do some data preprocessing. For almost all our datasets, no attribute values are missing, and for those that are missing attribute values, relatively few instances are missing values. Thus, to ensure all instances in the dataset have complete values, we will **remove all instances with missing attribute values**. The next phase of preprocessing will be for the categorical features, which will be dealt with by applying **one-hot-encoding**. For numerical variables, **z-score normalization** will be applied to ensure all values are between 0 and 1. It transforms the features, so they have the properties of a normal distribution with a mean of 0 and a standard deviation of 1. The equation is shown here:

$$z = \frac{x - \mu}{\sigma}$$

It is also required to implement a **multi-layer feedforward network** with **backpropagation learning** capable of training a network with an arbitrary number of inputs, hidden layers, hidden nodes by layer, and number of outputs.300 The backpropagation algorithm trains the network by iteratively adjusting the weights to minimize the error between predicted and actual outputs, using gradient descent. The error is computed through a **forward pass**, which involves propagating the input data through each layer, where the individual nodes compute weighted sums, add biases and apply activation functions. The **backward pass** is used to update the individual weights based on the error gradient.

The nodes will be laid out in layers, every node in each layer will connect to every node in the next layer. The hidden nodes will incorporate a **sigmoid activation function** with either **logistic tangent**

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

or **hyperbolic tangent**.

$$\sigma(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

The output node(s) will depend upon what problem the network is being used for: For regression, a **linear activation function** will be applied at the output; for classification, the **softmax activation** function will be used (multinomial class distribution where all classes are treated equally). The softmax activation function is shown below:

$$P(C_i|x) = \frac{exp\ (w_i{}^T x\ +\ w_{i0})}{\sum_{j=1}^{K}\ \ exp\ (w_j{}^T x\ +\ w_{j0})}$$

We will then formulate a **hypothesis** focusing on convergence rate and final performance of each of the different network architectures for each of the various problems.

The backpropagation algorithm is **tested on networks with 0, 1, and 2 hidden layers**. We use **ten-fold cross-validation** to train on nine of the partitions and test on the remaining partition. After rotating 10 times through the partitions and every partition was used exactly once as a test set, an average of the performance is computed. We will compare these results using the metrics of **precision, recall, accuracy, and mean squared error**. Comparing these results, we will be able to **evaluate our hypothesis**.

Implementing the feedforward network involves the tuning of numerous hyperparameters, such as the number of nodes in each layer, the learning rate, the minimum momentum, and the maximum number of training iterations before terminating the training process. **Hyperparameter tuning** is done by extracting 10% of the data (random, but stratified). Then the remaining 90% are split into ten folds of 9%. For each of the experiments, nine of the folds are combined, holding out the tenth as our test set. Tuning is done by training 10 times, once on each of the nine fold sets while tuning with the initial 10%. The hyperparameters are chosen by selecting the best of the averaged results. The **generalization ability** is tested 10 times, one for each unique held-out fold.

---

**System Architecture**

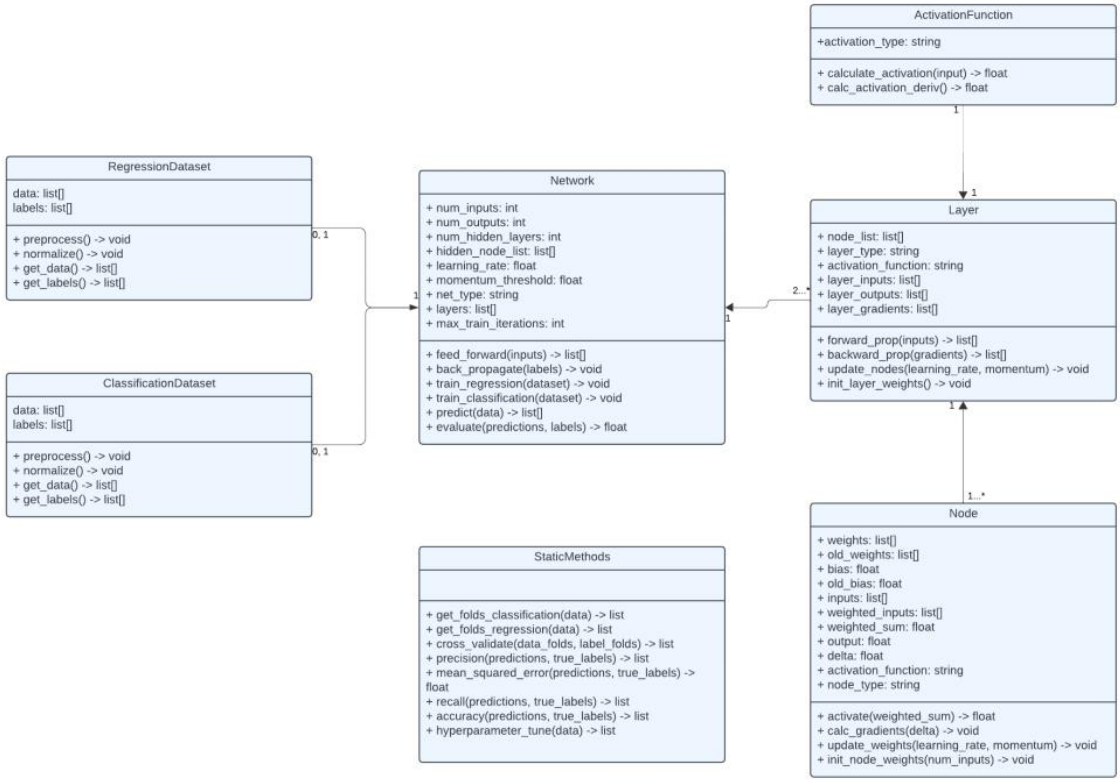


*Figure 1. UML Class Diagram for Feed Forward Neural Network*

Network Class

- This class contains instances of Layer classes which contain all the nodes. This class is meant to hold all the big picture methods that call helper methods in smaller classes. This class is also where you will take in the dataset classes.
  The attributes for this class are:
  1. num_inputs: This is an int which will represent the number of input nodes the network will take.
  2. num_outputs: This is an int which will represent the number of output nodes the network will have.
  3. num_hidden_layers: This is an int which will represent the number of hidden layers in the network.
  4. hidden_node_list: This is a list of ints which represents the number of nodes in each hidden layer. The length of this list should be the same as num_hidden_layers.
  5. learning_rate: This is a float that will represent the learning rate of the network.
  6. momentum_threshold: This is a float that will represent the momentum threshold of the network.
  7. net_type: This is a string which will represent if the network is a classification network or a regression network.
  8. layers: This is a list of Layer classes that will contain each specific layer that the network contains.
  9. max_train_iterations: This is an int that represents the max amount of training iterations the network will complete before stopping to prevent an infinite loop and overfitting.

  The functions for this class are:
  1. feed_forward(): This function takes in a list of inputs from the input layer of the network and it will call the feed forward method in the layer class. It will then update the nodes required in each layer.
  2. back_propagate(): This function will take in a list of targets and it will call the backward_prop method in the Layer class. This will then update each node's instance variables.
  3. train_regression(): This function will take in a regression dataset class, and it will train the network to predict values based off of the training data.
  4. train_classification(): This function will take in a classification dataset class, and it will train the network to predict specific classes based off of the training data.
  5. predict(): This function will take in a list of data points and it will predict each datapoint and return a list of predictions.
  6. evaluate(): This function will take a list of predictions and true labels. It will then calculate metrics for the model and return them.

Layer Class

- This class represents a layer of a network. It contains an arbitrary number of nodes and it is responsible for calling all the helper methods in the Node class to complete the overall goal of the network.

  The attributes for this class are:
  1. node_list: This is a list of Node classes that represents all the nodes in a given layer.
  2. layer_type: This is a string that represents if the layer is an input layer, hidden layer, or output layer.
  3. activation_function: This is a string that represents the type of activation function for the layer.
  4. layer_inputs: This is a list of all the inputs for a layer from the previous layer.

5. layer_outputs: This is a list of all the outputs for a layer going to the next layer.
6. layer_gradients: This is a list of all the gradients for a layer for all the corresponding nodes calculated from backpropagation.

The functions for this class are:

1. forward_prop(): This function takes in a list of inputs and applies the weights, biases, and activation.
2. backward_prop(): This function takes in the gradients from the next layer and it calculates all the gradients for the previous layer. It then returns these gradients.
3. init_layer_weights(): This function calls init_node_weights for all the nodes to set up the layer correctly.
4. update_nodes(): This function updates all the node weights inside the layer by calling forward_prop, and backward_prop.

Node Class

- This class represents each of the specific nodes in a layer. These nodes contain weights and biases as well as other book keeping variables.

The attributes of this class are:

1. weights: This is a list of all the weight connections from the previous layer
2. old_weights: This is a list of all the old weights before they get updated. This is used to calculate the momentum term to test against the momentum threshold. This will then be used to decide if training can terminate.
3. bias: This is a float that represents the bias value at a specific node.
4. old_bias: This is a float value that represents the old bias value at a specific node before it gets updated. This is used to calculate the momentum term to test against the momentum threshold. This will then be used to decide if training can terminate.
5. inputs: This is a list of all the inputs from the nodes at the previous layer.
6. weighted_inputs: This is a list of all the inputs from the previous layer with the weights applied.
7. weighted_sum: This is a float that represents the sum of all the weights.
8. output: This is a float that represents the output of the node after all the weights and biases have been applied.
9. delta: This is a float that represents the error term in the node.
10. activation_function: This is a string that represents the activation function that should be used at this specific node.
11. node_type: This is a string that represents the type of node that the specific node is (input, output, hidden, etc.).

The functions for this class are:

1. activate(): This function takes in a weighted sum and applies the activation function to it. It then returns this value.
2. calc_gradients(): This function takes in the delta term and returns a list of calculated gradients.
3. update_weights(): This function takes in a learning rate as well as a momentum and then it will update all the weights in the specific node.
4. init_node_weights(): This function takes in a number of inputs then it will randomly set up all the weights for those edges.

ActivationFunction Class

- This class is where the activation function for a node will be held. This class contains the functions to calculate the activation value.

  The attributes for this class are:
  1. activation_type: This is a string that represents the type of activation function that will be used.

  The functions for this class are:
  1. calculate_activation(): This function takes in an input float value and returns an "activated" value.
  2. calc_activation_deriv(): This function returns the derivative of the activation function that is being used.

RegressionDataset Class

- This class is where all the data from a regression dataset will be held. It will do all the necessary things to preprocess the data before it gets handed to the network.

  The attributes for this class are:
  1. data: This is a list that will contain all the data after preprocessing without any labels
  2. labels: This is a list that will contain all the labels for the data.

  The functions for this class are:
  1. preprocess(): This function will preprocess all the data. This could be removing data points with missing values or one-hot-coding.
  2. normalize(): This function will normalize all the data as specified above in the requirements.

ClassificationDataset Class

- This class is where all the data from a classification dataset will be held. It will do all the necessary things to preprocess the data before it gets handed to the network.

  The attributes for this class are:
  1. data: This is a list that will contain all the data after preprocessing without any labels
  2. labels: This is a list that will contain all the labels for the data.

  The functions for this class are:
  1. preprocess(): This function will preprocess all the data. This could be removing data points with missing values or one-hot-coding.
  2. normalize(): This function will normalize all the data as specified above in the requirements.

StaticMethods

- This contains all of the helper functions that may be needed to calculate things that are not directly related to the main algorithm or datasets.

  1. recall(): This function takes in the predicted labels and the true labels and returns the recall value.
  2. precision(): This function takes in the predicted labels and the true labels and returns the precision value.

3. accuracy(): This function takes in the predicted labels and the true labels and returns the raw accuracy value.
4. mean_squared_error(): This function takes in a list of predicted values and a list of true values. It then returns the calculated mean squared error.
5. get_folds_classification(): This function takes in a set of classification data and returns a list of ten specific folds. The process of stratifying across folds happens in this function.
6. get_folds_regression(): This function takes in a set of regression data and it will sort the data based on the response value in the data. It will then break the data into groups. Then it will draw the first item from each group for the first dataset and the second for the second dataset.
7. cross_validate(): This function takes in a set of folds and it will perform 10-fold cross-validation. This function will return a list of metrics to evaluate the model.
8. hyperparameter_tune(): This function will take in a set of data and tune all the hyperparameters in the model (k, sigma, etc.). It will then return the parameters that performed best on average.

---

**System Flow**

To run the system and make sure that all the requirements are fulfilled, start by creating all of the datasets that are described in the requirements. This is where the flow of the data will start.

Next, you will need to set up the network class. To do this you will need to set up all of the hyperparameters for the specific task you are trying to complete. These hyperparameters are the amount of nodes that are in each layer, the learning rate, minimum momentum, and the max amount of training loop iterations that the model will do before terminating training. You will also need to specify if the network is a classification network or a regression network. These hyperparameters will be tuned via the hyperparameter tune function. We will use 10% of the original data as a hold out fold to test the different hyperparameters on. We will then use the hold out fold as our testing fold when using 10-fold cross validation, ignoring the fold that would normally be used as the testing fold. We will then decide which parameters to use by performing a basic grid search and using the combination of parameters that performed best based off of evaluation metrics.

Once the hyperparameter tuning is done, you will then be able to move into the actual training and testing of the model. To do this, use the data separate from the tuning hold out fold, and use that as your entire new dataset.

Next, use the new dataset as a parameter in the train regression function or the train classification function depending on what type of network you are training. These functions will then use the data passed in as a parameter to call the backpropagate and feed forward functions. These functions then use different parts of the data to follow different algorithms to set all of the weights and biases on each and every node. Once the weights and biases of each node are set, you will be able to predict data points. The way that we will set up our training and testing is by splitting them up into stratified folds. This will be done by having an equal amount of each class in each fold. We will then use 10-fold cross validation again to test our final model with 0, 1, and 2 hidden layers. We will then use these results to compare the models. You can follow the steps above to test every dataset. All you will need to change is the network type in the network initialization depending on a classification or regression dataset. By following these steps you should complete all the requirements that are listed in the requirements section.

---

**Test Strategy**

To ensure that we meet project requirements, we will first validate that our data has been effectively preprocessed. This means that our datasets contain only **complete instances** and our categorical values have been **encoded to binary values**. Because there are few incomplete instances to be removed, we will validate that they have been removed properly by simply outputting a message with the instances removed and comparing it to the instances we can see need to be removed in the data file. To validate that categorical values are properly encoded to binary values, we will test if our binary encoding is working as intended on some sample values and ensure that our algorithm is getting reasonable results (improperly encoded values would drastically alter the results of our experiment).

To test that our **normalization function** is working as intended, we will validate that it is applied in all situations stated in the design document, and test that it produces the correct values on some sample values that could reasonably be encountered. Further, we can test that all numerical values are between 0 and 1 after preprocessing is complete and normalization has been applied. These test cases together should be exhaustive enough to ensure that we are using normalization as intended.

To test that our **activation function, feed-forward neural network, and backpropagation** work as intended, we will run some sample iterations by using sample instances from our various datasets on their respective neural networks, outputting the weights of each node connection, the values in each weighted sum for each node in the hidden layers and output layer, the weighted sums from those values, and the result of the activation function given that sum. Then, we will print out the errors, the gradient, and the changes made to the weights of the connections between nodes. This way, we will be able to validate that all values are calculated properly and that adjustments to the weights are properly made based on those values. We will run these tests for an exhaustive selection of samples, including both regression and classification problems, so that we can fully validate that the algorithm works.

To test that our **grid search hyperparameter tuning** is working as intended, we will print values of the hyperparameters as we run the algorithm, checking to see that it changes values, moving through the whole grid. Further, we will ensure that the hyperparameter values that enable the best performance are selected by printing the performance of the model after training for each possible set of hyperparameter values.

Upon obtaining our results, we will check to make sure that **precision, accuracy, recall, and F1 score** have been calculated correctly by printing the intermediate values used to calculate these metrics and carrying out the calculations manually.

We will check that **10-fold cross-validation** works as intended by ensuring that we get different results on different trials of the experiment and that we get reasonable results, that we run 10 trials of tuning using different folds for each run, and that the average performances are calculated properly.

---

**Task Assignments and Schedule**

Throughout the course of the project, each team member is expected to complete their work that is assigned in a timely manner. On the parts of the project that require team members to meet in person, it is expected that the team members will communicate to make a time work where they can meet and complete the task at hand. This communication can be done directly with the other team members since there are only three members in the team. All of the code that is written in this project will be uploaded to a team repository so that it can be accessed at any time if the other team members need it. It is expected that team members will keep their code up to date in the repository if they have made any changes.

Task Descriptions

1. Create Design Document - Follow the Creating a Design Document PDF and on D2L to create a design document specific to project 3.
2. Create Hypothesis - Using the descriptions of the datasets and of the neural network algorithms, create a hypothesis about how the different datasets will perform according to the chosen metrics.
3. Update dataset classes' methods - Using classes from project 2 as a template, implement any new methods needed for the Neural Networks according to the description of its methods in the Project 2 design document. Specifically, implement new preprocessing methods, primarily normalization. After this task, these classes should be ready for use.
4. Create the Node class - Follow the description of the Node class in the Project 3 Design Document. At this stage, the class only needs its structure: methods do not need to be implemented.
5. Implement methods for the Node class - Follow the method descriptions for the Node class in the Project 3 Design Document. After this task is complete, the class should be fully utilizable.
6. Create the Layer class - Follow the description of the Layer class in the Project 3 Design Document. At this stage, the class only needs its structure: methods do not need to be implemented.
7. Implement methods for the Layer class - Follow the method descriptions for the Layer class in the Project 3 Design Document. After this task is complete, the class should be fully utilizable.
8. Create the ActivationFunction class - Follow the description of the ActivationFunction class in the Project 3 Design Document. At this stage, the class only needs its structure: methods do not need to be implemented.
9. Implement methods for the ActivationFunction class - Follow the method descriptions for the ActivationFunction class in the Project 3 Design Document. After this task is complete, the class should be fully utilizable.
10. Create the Network class - Follow the description of the Network class in the Project 3 Design Document. At this stage, the class only needs its structure: methods do not need to be implemented.
11. Implement methods for the Network class - Follow the method descriptions for the Network class in the Project 3 Design Document. After this task is complete, the class should be fully utilizable.
12. Update StaticMethods - Using the StaticMethods from project 2, implement any new methods needed for methods needed for the Neural Network according to the description of its methods in the Project 3 design document. After this task, these methods should be ready for use.
13. Final Code Check - For this task, the team members will need to meet in person to make sure all the code runs as expected so that the data that gets collected later in the project is reliable. Further, team members will check that all code is well-commented and readable
14. Collect Results and Create Figures - For this task, one team member will need to create methods for generating figures from the results of our experiment and run the code to generate those figures.
15. Create a Rough Draft of the Paper - Each team member will be responsible for different sections of the paper, using course materials, the hypothesis created in the beginning of the experiment, and the results generated at the end of the experiment to complete the paper's different sections.
16. Complete the Paper - Team members will review one another's work on the paper and ensure that it meets all standards laid out on the project description on D2L, addresses the hypothesis, and accurately interprets the results.
17. Plan Video - This task requires the general layout of the video to be complete and any "dummy methods" to be implemented. This allows the actual recording of the video to go much smoother and prevents last minute changes while trying to record.
18. Record Video - For this task, a team member will record the video. Another team member will edit the video if necessary. This process will also be used as a final check for anything missing in the project. After this task is complete, the paper, video, and code will be submitted


*Figure 2. Gantt chart for schedule of tasks on Neural Network project.*