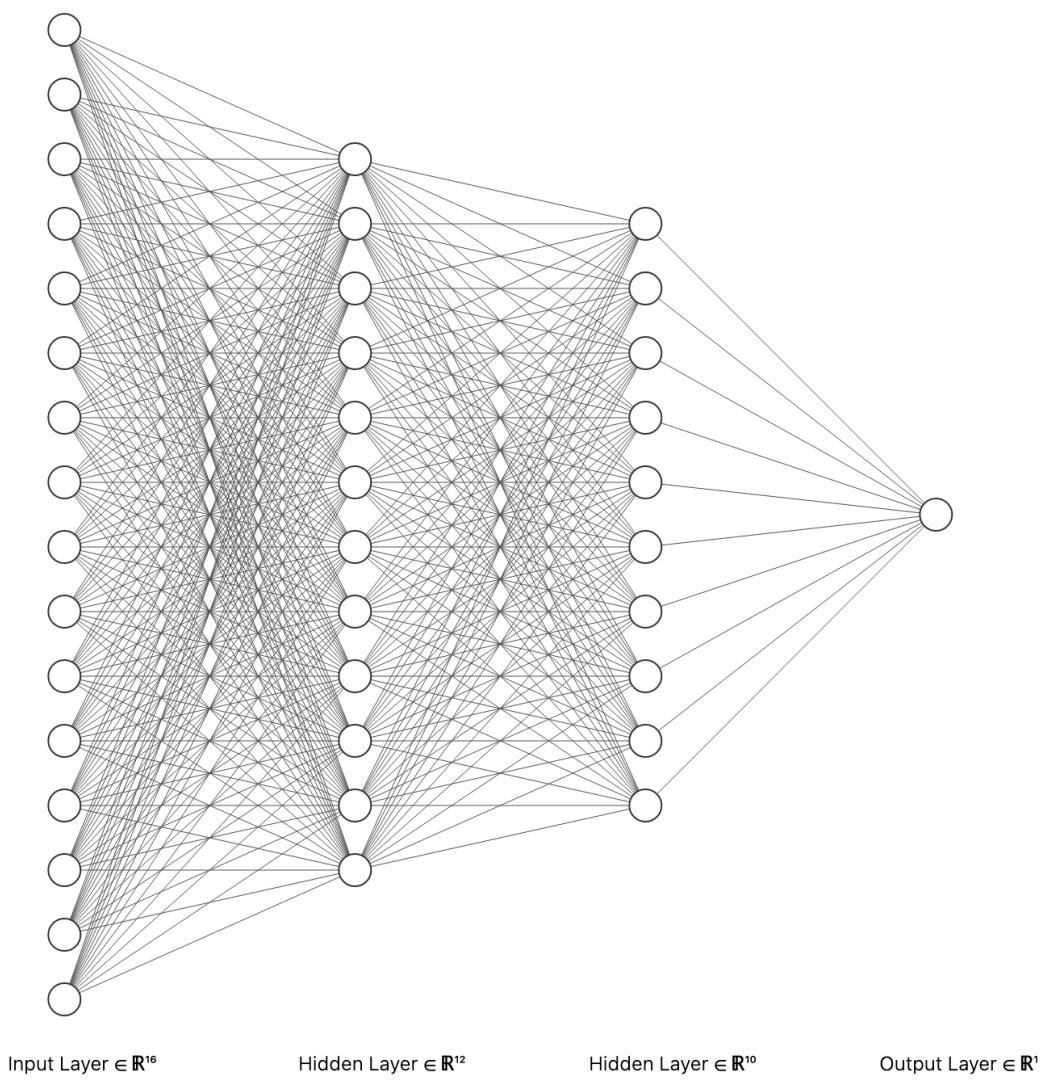


MultiLayer Perceptron Classifier

Nikolaos Laloutsos



PART A: Brief analysis and explanation of the code

The MLP classifier consists of a total of 4 classes, each located in a separate file. Leveraging the object-oriented features of the language, I created them hierarchically:

Neuron class → Layer class → MultiLayerPerceptron class → PerceptronClassifier class

1) *Neuron Class:*

- **Nueron(Constructor):** Creates an individual neuron as an object and initializes its attributes (input size, weights, gradient size, activation function, and its derivative). It also initializes the weights with random numbers as required by the assignment.
- **forward:** Calculates the output of the neuron. It simply takes the sum of inputs × weights and then applies the activation function to this sum.
- **Backward:** Calculates the error ($\delta = g'(u) * \text{errorTerm}$) needed by the neurons of the previous layer and returns it, so that the neurons in the previous layer can use it. It also updates its own attributes wherever necessary.
- **Update:** Applies the gradient descent step to the neuron and updates its weights.
- **zeroGradients:** Resets the neuron's gradients to zero. This is done after completing each batch.

2) *Layer class:*

- **Layer(Constructor):** Creates a new hidden layer. Given the number of neurons, input size, activation function, and its derivative, it generates a list of neuron objects, representing the neurons of that specific layer.
- **layerForward:** Iterates through each neuron in the layer and computes its output (executes `neuron.forward`). It then stores the outputs in an array.
- **layerBackward:** Exactly following the same logic as above, it performs backward propagation for each neuron in the layer.
- **layerUpdate:** Updates the weights of each neuron, a method that has already been implemented.
- **zeroGradients:** Resets the gradients of each neuron in the layer.

3) *MultiLayerPerceptron Class:*

- **MultiLayerPerceptron(Constructor):** It creates a new neural network. Given a list of numbers as a field, the number of elements determines the number of hidden layers, and the values of these elements indicate how many neurons each layer will contain. **Important:** The output layer and the activation function are created separately at the end, outside of the constructor. Naturally, these are specified by the user
- **forwardPass και backwardPass:** As before, it now traverses all layers and transfers the outputs, either from front to back (**backward**) or from back to front (**forward**).
- **train:** The main and most important method of the class executes the **Gradient Descent** algorithm with **Backpropagation**. Specifically, for each **epoch**, it splits the input data X and the corresponding target outputs T into **batches**, performs a **forward pass** to calculate predictions, computes the **MSE** (Mean Squared Error) and the partial derivative of the error with respect to the output, and then performs a **backward pass** to update the weights of the neurons. Weights are updated based on the **learning rate**, adjusted for the batch size. The process continues until it reaches the maximum number of epochs or until the change in error between epochs falls below a defined **errorThreshold**. At the end of each epoch, it prints the MSE to monitor the training progress.
- **test:** It performs a **forward pass** using the **test set** as input. It then calculates the percentage of correct predictions (accuracy). It returns a list containing all predictions, so that we can save it to a **CSV** file if desired.

4) *Perceptron_Classifier:* The **Main** method of the program. The user inputs all parameters to define both the architecture and the learning process of the neural network. It also reads the data, stores them in the necessary lists for **training** and **testing**, and encodes them as demonstrated in class (e.g., **One-Hot Encoding**: C1 - > 1,0,0,0, etc.). Next, it informs the user via the command line about the upcoming operations based on their choices, asks for confirmation to proceed, and executes the training. Once training is complete, it performs a **forward pass** on the test set. Depending on which of the four values is closest to 1, it classifies the input into the corresponding category. Finally, it prompts the user to choose whether to save the results.

PART B: Execution Instructions.

After the program is compiled (*javac .java), we run it using the command '**java Perceptron_Classifier**'. Then, using the '**define**' command, we specify all the parameters required for the network to run:

define(number of batches, max epochs, error threshold, learning rate, input size, number of categories, number of neurons, activation function, number of neurons, activation function... , ... , output activation function)

Note: We can add as many hidden layers as we want + the output layer; for the requirements of this assignment, 3 layers are used.

- Some indicative commands for direct copy-paste into the command prompt:

```
define(8,2000,0.0000001,0.01,2,4,12,tanh(u),12,tanh(u),12,tanh(u),sigmoid)
```

```
define(1,2000,0.0000001,0.01,2,4,12,tanh(u),12,tanh(u),12,tanh(u),tanh(u))
```

```
define(16,2000,0.0000001,0.01,2,4,6,tanh(u),4,tanh(u),3,relu,sigmoid)
```

```
define(4000,2000,0.0000001,0.01,2,4,12,tanh(u),12,tanh(u),12,relu,sigmoid)
```

```
define(32,2000,0.0000001,0.01,2,4,16,tanh(u),12,tanh(u),8,tanh(u),sigmoid)
```

Afterward, the program prints the characteristics of the model to be examined and evaluated to the console and asks if we wish to proceed. Once it finishes, it asks the user whether they want to save all the model's predictions in relation to the actual values to a CSV file

PART C: Search and Analysis of the models and the results.

In addition to the main program where you can manually create, train, and test a model, I developed another Java file that performs a **Random Search** for all parameters except for the number of layers. Specifically, it trains and then evaluates each model. At the end, it saves the characteristics and predictions of the best-performing model into one CSV file, and the characteristics and accuracy of all tested models into another. I ran this for 200 experiments and kept the one with the highest score on the test set.

The idea for Random Search came from my university **Machine Learning** course, as well as its frequent use in various **Kaggle** competitions. The parameter space was far too large to perform a **Grid Search**. Out of the 200 randomly generated and tested models, the best one achieved an accuracy of **96.65%** on the test set.

- Many models that appeared likely to achieve low training error did not have enough time to do so, as they required an excessive number of **epochs**.
- Note: I capped the number of neurons per layer at 20, as manual experimentation revealed that larger layers led to increased complexity without any significant gain in accuracy.

- For the **learning rate**, it chose randomly between 0.001 and 0.0001, while for the **batch size**, it had the following options::

```
int batch_size = new int[]{8, 16, 32, 64,
128, 256, 1000, 2000, 3000, 4000};
```

Thus, the best found classifier is:

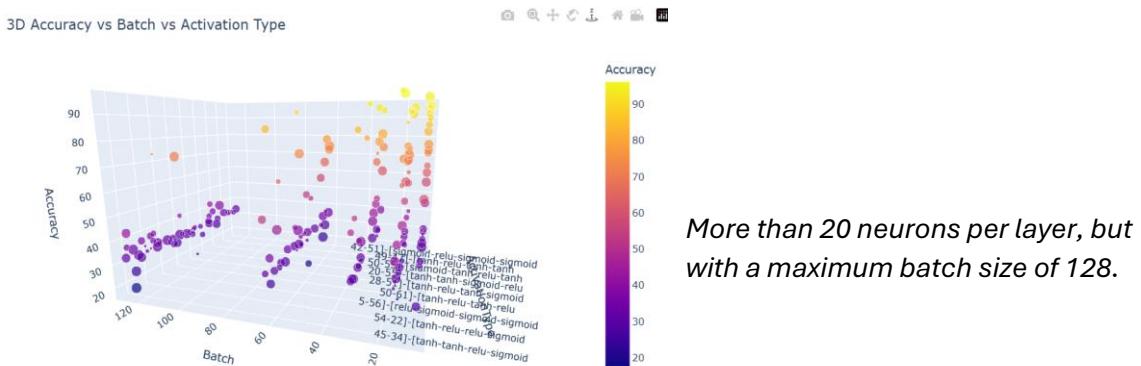
== BEST CLASSIFIER PARAMETERS ==				
Accuracy	96.65			
Layers	[13	11	15]	
Activations	tanh	tanh	tanh	sigmoid]
Batch	32			
Epochs	3730			
LearningRate	0.00782			
ErrorThres	1.00E-07			

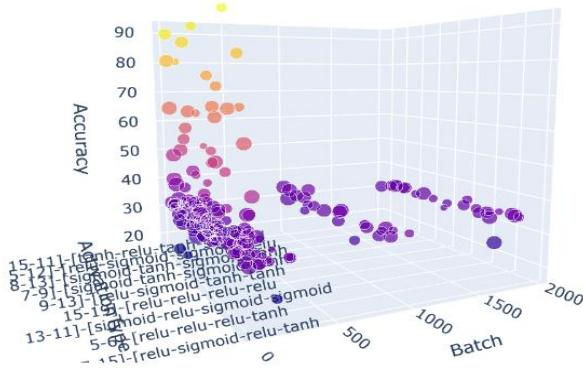
Subsequently, I created a test set with 10,000 random samples, where the aforementioned classifier achieved 91% accuracy. Meanwhile, another classifier*—which had a 94% accuracy rate on the original test set—maintained the same consistent accuracy on the 10,000-sample set, performing better than the first one. Ultimately, this model proved to be more stable and reliable than the initial 'winner' among the 200 different models.

- **define(8, 2000, 0.0000001, 0.01, 2, 4, 12, tanh(u), 12, tanh(u), 12, tanh(u), sigmoid)**

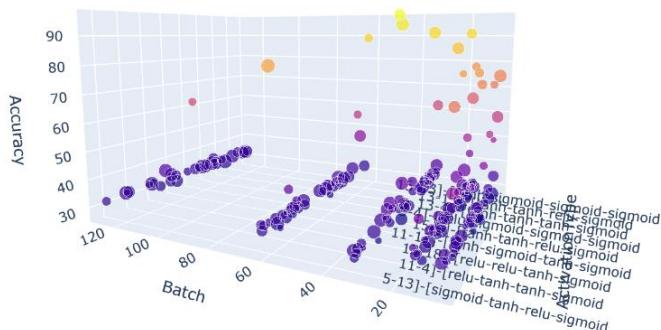
I observed that with high **batch sizes**, the model did not achieve good accuracy; therefore, I repeated the **Random Search** with a maximum batch size of 128, aiming to potentially find an even better model (none was found, though the overall quality of models improved). Moving forward, I wanted to examine the relationship between batch size, the combination of **activation functions**, and **generalization** capability.

The result was the following 3D plot (**Activation Functions vs. Batch Size vs. Accuracy**), generated via Python by reading the CSV file containing the Random Search results:





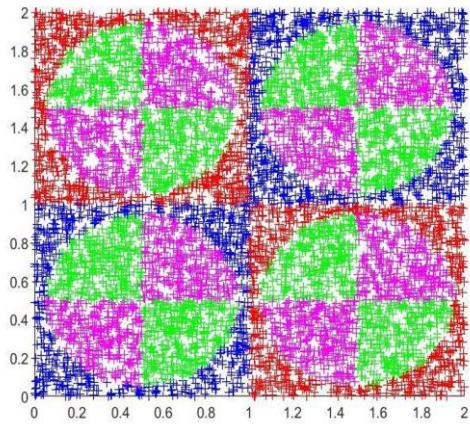
Max batch size = 4000 but 20 or less neurons per Layer



Max batch size = 128, 20 or less neurons per layer.

The smaller the **batch size**, the better the models performed. When a large number of neurons was permitted, there was at least one high-performing model for almost every combination of **activation functions**. It is noteworthy that across 200 experiments, **Random Search** failed to find a model with good **generalization** that was trained with a batch size greater than 500. All the 'good' models had a **training error** between 0.01 and 0.005 by the end of their training.

Error Analysis:



By examining the points where even the models with the best **generalization** capabilities failed, I noticed that errors occur around specific regions.

Specifically, errors occur at the center of the virtual circles in the plot and at the outer boundaries of the circle, where the categories seem to 'come into contact from all directions.' For instance, the best classifier found via **Random Search** had errors at the coordinates (0.06, 1.4) and (0.44, 1.47).