Institute of Technological Studies of Bizerte
Department of Electrical Engineering

# Lab Report

Nourhen Mendili
*nourhenmendili8@gmail.com*

Rania Maghroum
*raniamaghroum7@gmail.com*

*Abstract* — **The main topics discussed in the manuscript.**

## *Introduction:*

This report explores the implementation of a Multilayer Perceptron (MLP) in Julia to solve a classification problem using the Iris dataset. Each step of the script will be explained in detail, including data preprocessing, model configuration, training, performance evaluation, and result visualization. The objective is to provide a clear understanding of the concepts and techniques used to develop this machine learning system.

## *Setup and Data Preparation:*

### *A. Initialization and library Imports*

```
using Printf
```

Define the display method for floating-point numbers so that they are shown with three digits after the decimal point

IO represents the input/output object on which the display will be made

```
Base.show(io::IO, f::Float64) = @printf(io, "%1.3f", f)
```

Import the module for operations related to random numbers.

```
using Random
```

Set a seed to ensure the reproducibility of random results.

```
Random.seed!(1234)
```

Add the src directory from the current directory (pwd()) to the module search path to allow importing custom modules.

```
push!(LOAD_PATH, pwd() * "/src")
```

Allows for automatically reloading modifications made to the code without restarting the Julia session.

```
using Revise
```

Imports modules for data processing.

```
using Preprocessing
```

Provides activation functions used in neural networks, and creates and manipulates Multilayer Perceptrons (MLP), a type of neural network consisting of fully connected layers.

```
using ActivationFunctions, MLP
```

Used to evaluate the performance of machine learning models.

```
using Metrics
```

### *B. Hyperparameters*

Define a mutable structure for hyperparameters:

mutable struct Settings

Number of epochs

```
epochs::Int
```

Batch size used during training.

```
batch_size::Int
```

Checks that batch_size is valid ($\geq 1$).

```
Settings(epochs, batch_size) = batch_size ≥ 1 ? new(epochs, batch_size) : error("Batch
size must be greater than 1")

    Settings(epochs) = new(epochs, 1)

end
```

Create an instance of Settings with 20 epochs and a batch size of 16. de 16.

```
hp = Settings(20, 16)
```

### C. Dataset Preparation

Load predefined datasets.

```
using RDatasets
```

Load the iris dataset from the datasets library.

```
iris = dataset("datasets", "iris")
```

Data Preprocessing Extract the features (x) and convert them to a Julia array.

```
x = iris[1:end, 1:end-1] |> Array
```

Calculate the number of features (num_features).

```
num_features = size(x)[2]
```

Use map to replace class names with numbers: setosa → 1, versicolor → 2, virginica → 3.

```
vspecies = map(x -> if x=="setosa" x=1 elseif x=="versicolor" x=2 elseif x=="virginica"
x=3 end, iris.Species)
```

Determine the total number of classes (num_targets).

```
num_targets = maximum(levels(species))
```

Create a One-Hot Encoding Matrix for Labels Create a matrix y where each row corresponds to an example, and each column represents a class.

```
y = zeros(length(species), num_targets)
```

Fill y by encoding the labels in a one-hot format

```
y[species .== 1, 1] .= 1

y[species .== 2, 2] .= 1

y[species .== 3, 3] .= 1
```

Split the Data into Training, Testing, and Validation Sets Divide the data into three subsets: 70% for training (train_size=0.7), 10% for validation (val_size=0.1), and the rest for testing.

```
(x_train, y_train), (x_test, y_test), (x_val, y_val) = data_split(x, y, train_size=.7,
val_size=.1)
```

Create Data Loaders data_loader splits the data into batches of the defined size (batch_size=16)

```
data_x = data_loader(x_train, hp.batch_size)

data_y = data_loader(y_train, hp.batch_size)
```

## *Model Definition and Training:*

### *A. Define a Neural Network Architecture:*

Create a list of layers for the neural network.

```
model = [  MLP
```

Input feature number → 40 neurons, with ReLU activation function.

```
Layer(num_features, 40, relu; distribution='n'),
```

40 neurons → number of target classes, with softmax for probabilistic classification.

```
Layer(40, num_targets, softmax, distribution='n') ]
```

### *B. Parameter Configuration (regularization and optimizer):*

Regularization is a technique used to improve a model’s generalization and prevent overfitting by adding a penalty to its loss function. allows configuration of regularization, but since the method is set to :none, no regularization is applied, and the parameters have no effect.

```
reg = Regularization(:none, .2, .6, .0)  # method, λ, r, dropout
```

Using the cross-entropy loss function and stochastic gradient descent method, with a learning rate of 0.03 and a regularization configuration (without an active method).

```
solver = Solver(:crossentropy, :sgd, .03, reg)
```

### *C. Training Loop and Metrics Calculation:*

Variables to Store Losses

```
ltrn, ltst = [], []
```

This is part of the training process of a neural network model.

Main Epoch Loop

```
for epoch in 1:hp.epochs
```

Display the current epoch

```
printstyled("=================== EPOCH #$epoch =====================\n"; bold=true,
color=:red)
```

Training Loop on Data

```
for (data_in, data_out) in zip(data_x, data_y)
```

Handles all aspects of training, from data and model initialization to weight updates and performance evaluation

```
TrainNN(model, data_in, data_out, x_val, y_val; solver)
end
```

***>>Calculate Training Loss***

Apply the model to the training data to get predictions

```
ŷ_train = Predict(model, x_train)
```

Calculate the loss (or error) by comparing the model predictions (ŷ train) with the true values (y_train). The loss function (loss_fct) measures the deviation between these two values.

```
loss = loss_fct(y_train, ŷ_train; loss=solver.loss)
```

Add the calculated loss value to a list or data structure called

```
ltrn
```

push!(ltrn, loss)

***>>Calculate Test Loss***

Predict uses the neural network model to make predictions on a test dataset, x_test

```
ŷ_test = Predict(model, x_test)
```

Calculate the loss by comparing the true labels (y_test) with the model predictions (ŷ test)

```
loss = loss_fct(y_test, ŷ_test; loss=solver.loss)
```

Add the calculated loss value to a list or array

```
ltst
```

```
push!(ltst, loss)
```

Display End of Epoch Losses

```
printstyled("*** @ last *** "; bold=true, color=:green)

   println("train loss: $(ltrn[end]) *** test loss: $(ltst[end])")
end
```

***Evaluation and Visualization:***

***A. Plotting Loss Values***

```
using Plots
```

Create the Training Loss Plot

```
plot(ltrn, label="train", xlabel="epoch", ylabel="loss", title="loss values")
```

Add Test Loss to the Plot

```
p = plot!(ltst, label="test")
```

Display the Plot

```
display(p)
```

Use Predict to generate predictions from the model on the test set (x_test).

```
ŷ_tst = Predict(model, x_test)
```

Convert Predictions to Labels:

```
ŷ_tst = Int.(ŷ_tst .== maximum(ŷ_tst, dims=2))
```

## *B. Evaluation Metrics*

This is a table that presents the number of correct and incorrect predictions for each class.

```
cm(y_test, ŷ_tst)
```

Accuracy Score Calculates the accuracy score, which measures the proportion of correct predictions to the total number of predictions

```
accuracy_score(y_test, ŷ_tst);
```

F1-score Calculates the F1 score, which is the harmonic mean of precision and recall

```
f1_score(y_test, ŷ_tst);
```

***Conclusion:*** In this report, we explored the detailed implementation of a Multilayer Perceptron (MLP) in Julia to solve a classification problem using the Iris dataset. Each step, from data preprocessing to model configuration, training, evaluation, and result visualization, was analyzed and explained in depth.

Through this process, we demonstrated the practical application of machine learning concepts and techniques, highlighting how different components—such as activation functions, regularization methods, and optimization algorithms—interact to build an efficient and robust system.