Multilayer Perceptron Summary and Implementation

Brandon Marquez Salazar

I. Introduction

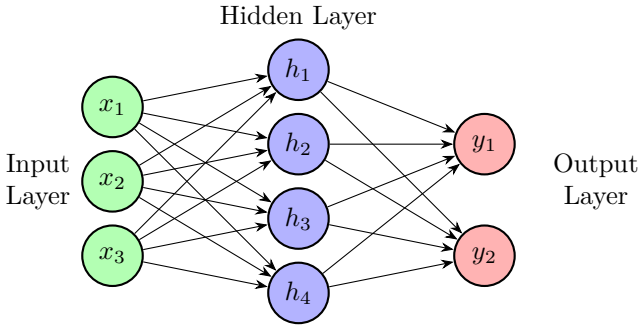
A perceptron is a simple neuron which computes a value and thresholds it through an activation function. This neuron has the ability to learn from any given input and it's expected output using a simple iterative process of training. This training process alters the coefficients of a weighed symbolic vector, which weights are used to compute a desired output and the values of the vector variables are the given inputs. This approach came from mathematical interpretation a brain cell's behavior, and its *tout seule* implementation is limited to linearly separable problems.

II. Artificial Neural Network and Multilayer Perceptron

An artificial neural network, ANN, is an elements net. Those elements (neurons) which are connected one to each other. Those nodes are organized in layers. There are three main types of layers:

- Input layer: This layer is the one that receives the input data.
- Hidden layer: The output of the neurons in this layer is not visible.
- Output layer: This layer outputs the final values.

A **Multilayer Perceptron** MLP is a static artificial neural network architecture where each node in a layer is connected to every node in the next layer.



III. Backpropagation Algorithm

Backpropagation is an algorithm used to train an ANN, this algorithm is a generalization of the gradient descent algorithm following the steps below:

1. Initialize the weights of the network at small random values.
2. Forward computation $\forall i$-th training pair $(x_i,\, o_i)$
   - $\forall n$-th neuron and each k-th neuron input, compute the input and output

$$v_{nk};\ o_{nk} = f(v_{nk})$$

3. Backward computation start computing from output layer $(L-th$ layer$)$
   - $\forall n$-th neuron at each $l-th$ layer, compute the error and delta at the moment $t-th$

$$\delta_n^L(t) = e_n^L(t) \cdot f'(v_n^L(t))$$
$$\delta_n^{l-1}(t) = e_n^{l-1}(t) f'(v_n^{l-1}(t))$$

   Where the error e at layer L and $l-1$ is defined by

$$e_n^L(t) = y_n^L(t) - o_n^L(t)$$
$$e_n^{l-1}(t) = \sum_{m=0}^{m_l} \delta_m^l(t) w_{mn}^l(t)$$

4. Weight update
   - $\forall n$-th neuron at each $r-th$ layer, update weights at the moment $t-th$

$$w_n^r(t+1) = w_n^r(t) + \Delta w_n^r(t)$$

   Considering that

$$\Delta w_{nk}^r = -\mu \sum_{i=0}^{N} \delta_n^l(i) y^{l-1}(i)$$

IV. MLP Implementation

In this section, a MLP will be implemented as a class in Python.

A. Installing and importing the required modules

For this implementation it's necessary to use the following modules: numpy for numerical operations, and matlplotlib for data visualization and scikit-learn for MLP implementation.

```
!pip install numpy
!pip install matplotlib
!pip install scikit-learn
```

```
import numpy as np
import matplotlib.pyplot as plt
import random as rd
## From scikit-learn, it's
# necessary to import the MLPClassifier class.
from sklearn.neural_network import MLPClassifier
## For MLP performance evaluation
from sklearn.model_selection \
  import train_test_split
from sklearn.model_selection \
  import cross_val_score
from sklearn.model_selection \
```

```
  import StratifiedKFold
from sklearn.metrics          \
  import accuracy_score
from sklearn.metrics          \
  import confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics          \
  import classification_report
## For datasets
from sklearn.datasets         \
  import load_iris
from sklearn.datasets         \
  import load_breast_cancer
from sklearn.datasets         \
  import load_wine
from sklearn.datasets         \
  import load_digits
from sklearn.model_selection \
  import train_test_split
from sklearn.preprocessing    \
  import StandardScaler

# Load datasets (to choose one)
def load_dataset(dataset_name='iris'):
    """Load a sklearn dataset and return X, y"""
    if dataset_name == 'iris':
        data = load_iris()
    elif dataset_name == 'cancer':
        data = load_breast_cancer()
    elif dataset_name == 'wine':
        data = load_wine()
    elif dataset_name == 'digits':
        data = load_digits()
    else:
        raise ValueError(
          "Choose: 'iris', 'cancer', 'wine', or 'd
        )
    return data.data, data.target
```

B. Defining the MLP class

```
class LovdogMLP:
    def __init__(
      self, layer_sizes=(10, 5),
      max_iter=1000, learning_rate=0.0001):
        ### Simple MLP class with sigmoid
        # activation function
        self.learning_rate = learning_rate
        self.layer_sizes = layer_sizes
        self.max_iter = max_iter
        self.model = None
        self.training_loss = []
        self.x = None
        self.y = None
        self.xTest = None
        self.yTest = None
        self.xTrain = None
        self.yTrain = None

    def setDataset(self, x, y):
        scaler = StandardScaler()
        self.x = scaler.fit_transform(x)
        self.y = y
        self.xTrain, self.xTest, \
        self.yTrain, self.yTest = \
          train_test_split(
            self.x,
            self.y,
            test_size=0.2
          )
        return self


    def create_model(self):
        # Create the Model
        self.model = MLPClassifier(
            hidden_layer_sizes=self.layer_sizes,
            activation='logistic',
            max_iter=self.max_iter,
            alpha=self.learning_rate,
            random_state=rd.randint(0, 1000),
            verbose=False
        )
        return self

    def train(self):
        if self.model is None:
            self.create_model()

        # Train and capture loss curve
        self.model.fit(self.xTrain, self.yTrain)
        self.training_loss =   \
          self.model.loss_curve_

        return self.training_loss

    def predict(self, x):
        if self.model is None:
            raise ValueError(
            "Model not trained yet. " +\
            "Call train() first.")
        return self.model.predict(x)

    def get_accuracy(self):
        predictions = self.predict(self.xTest)
        accuracy = np.mean(predictions == self.yTest)
        return accuracy

    def plot_training_loss(self,
      figsize=(8, 6), cmap="blues"):
        if not self.training_loss:
            print("No training data available")
            return
        colors = plt.get_cmap(cmap)\
          (np.linspace(
            0, 1, len(self.training_loss)
```

```
          ))

        plt.figure(figsize=figsize)
        plt.plot(self.training_loss,
          label='Training Loss',
          color=colors[0])
        plt.xlabel('Iteration')
        plt.ylabel('Loss')
        plt.title(
          f'MLP Training Loss (Layers: ' +
          f'{self.layer_sizes})')
        plt.legend()
        plt.grid(True)
        plt.show()

    def plot_confusion_matrix(self,
      figsize=(8, 6), cmap="Blues"):
        if self.model is None:
            raise ValueError("Model not trained yet")

        y_pred = self.model.predict(self.xTest)
        cm = confusion_matrix(self.yTest, y_pred)

        fig, ax = plt.subplots(figsize=figsize)
        ConfusionMatrixDisplay(cm)\
          .plot(ax=ax, cmap=cmap)
        ax.set_title(
          f"Confusion Matrix - MLP {self.layer_siz
        plt.show()
```

C. Definitions of different MLPs

Here 4 different MLPs will be implemented each one with 3, 4, 5 and 6 layers. All of them will be used to try to emulate logic gates functionalities.

```
mlp_of = {
  "3": LovdogMLP(layer_sizes=(20,10,10)      ,
      max_iter=1000),
  "4": LovdogMLP(layer_sizes=(20,20,20,10)   ,
      max_iter=1000),
  "5": LovdogMLP(layer_sizes=(20,40,40,20,10),
      max_iter=1000),
  "6": LovdogMLP(layer_sizes=(20,60,40,40,20),
      max_iter=1000)
}
## Color for each MLP and plette for plots and mat
cmpl = {
  "3":  "viridis" ,
  "4":  "magma" ,
  "5":  "plasma" ,
  "6":  "inferno"
}
```

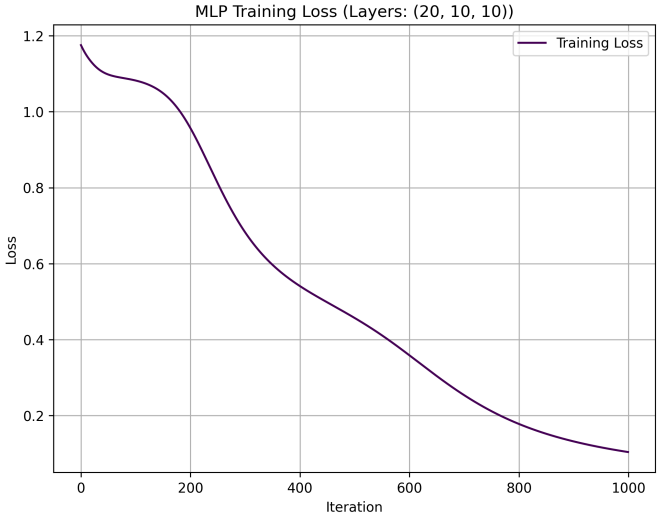
D. Testing the MLPs

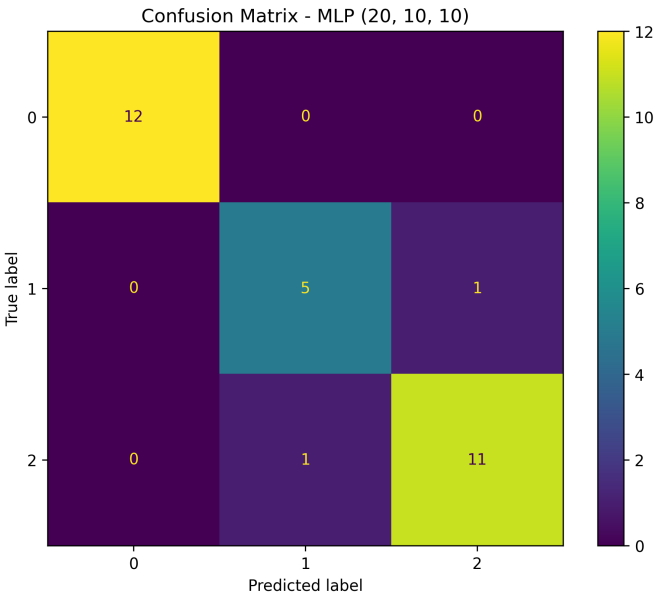
Now the MLPs will be tested to see if they can work with sklearn datasets.

```
dataset = {
  "x": load_dataset("iris")[0],
  "y": load_dataset("iris")[1]
}
for mlp in mlp_of:
    mlp_of[mlp].setDataset(
      dataset["x"], dataset["y"])
    mlp_of[mlp].train()
    print(f"Accuracy for {mlp} layers:"+
      f" {mlp_of[mlp].get_accuracy()}")
    mlp_of[mlp].plot_training_loss(cmap=cmpl[mlp])
    mlp_of[mlp].plot_confusion_matrix(cmap=cmpl[mlp])
    # Print Separator
    print("-"*40, end="\n\n")
```

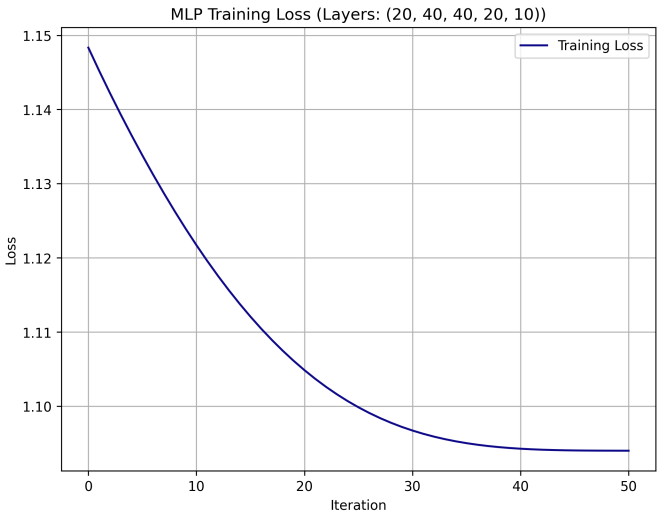
1) Iris Dataset:

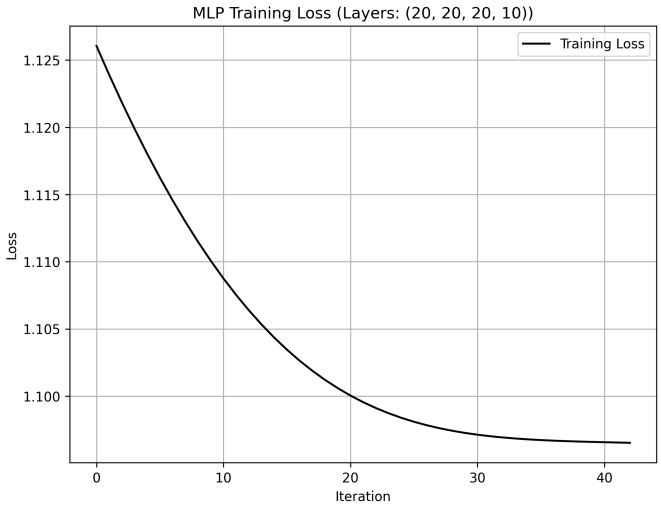
```
Accuracy for 3 layers: 0.9333333333333333
```

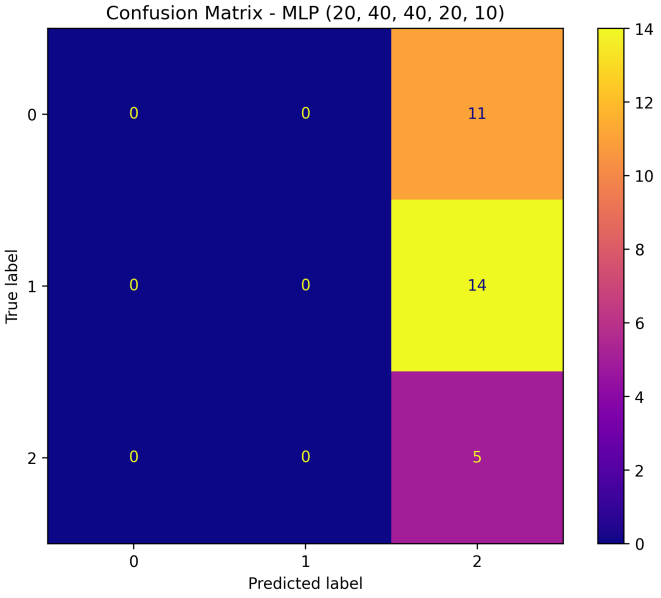


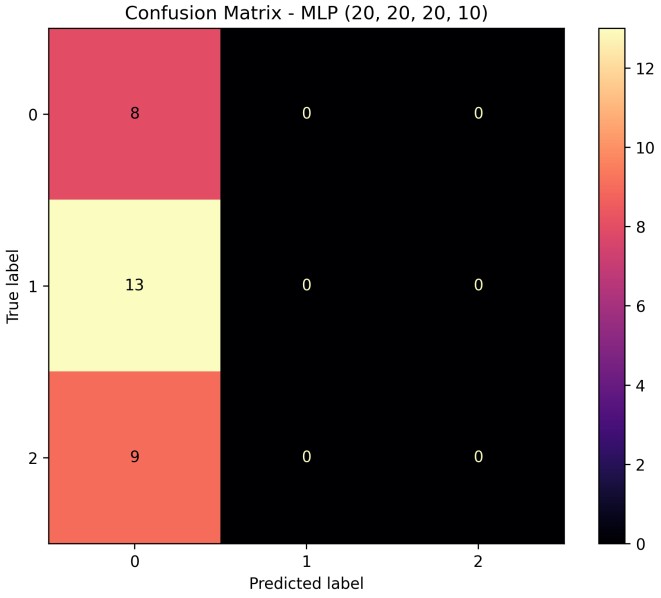


Accuracy for 4 layers: 0.26666666666666666

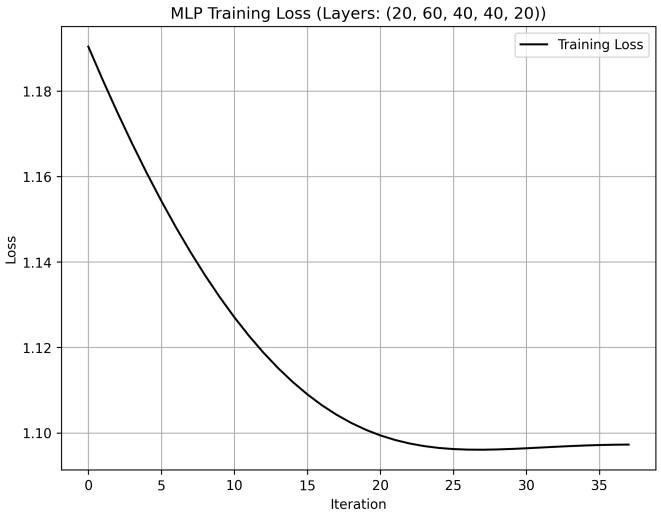




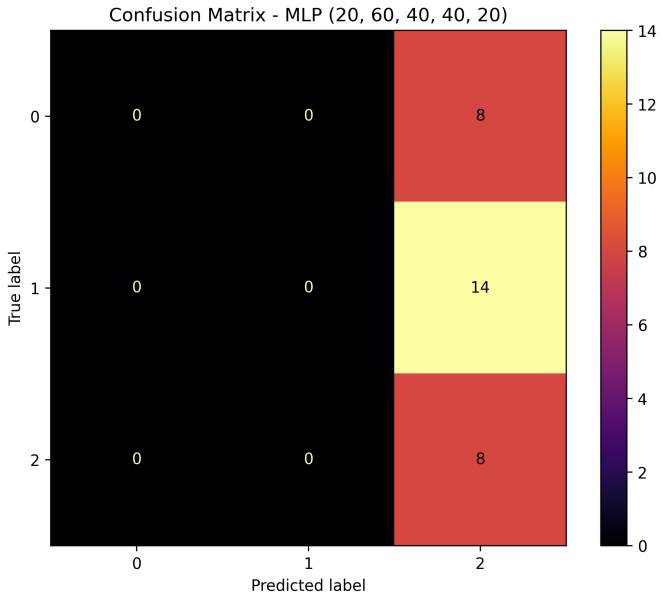


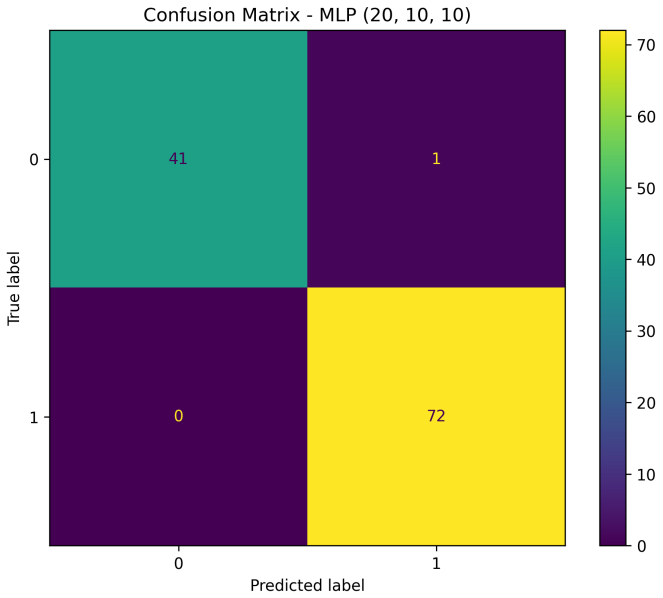


Accuracy for 6 layers: 0.26666666666666666



Accuracy for 5 layers: 0.16666666666666666



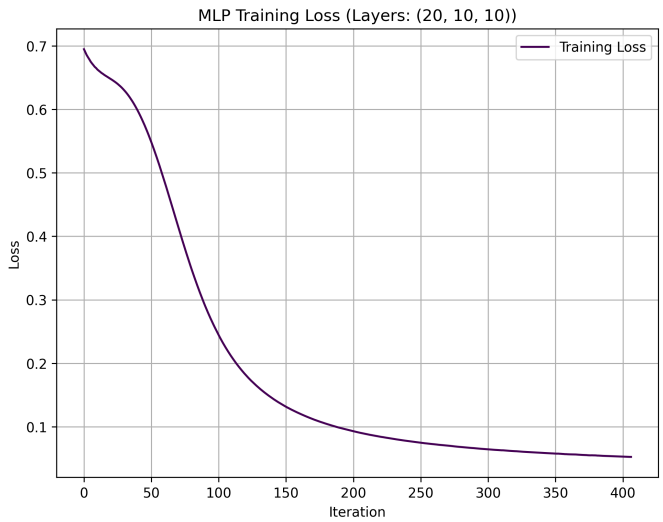


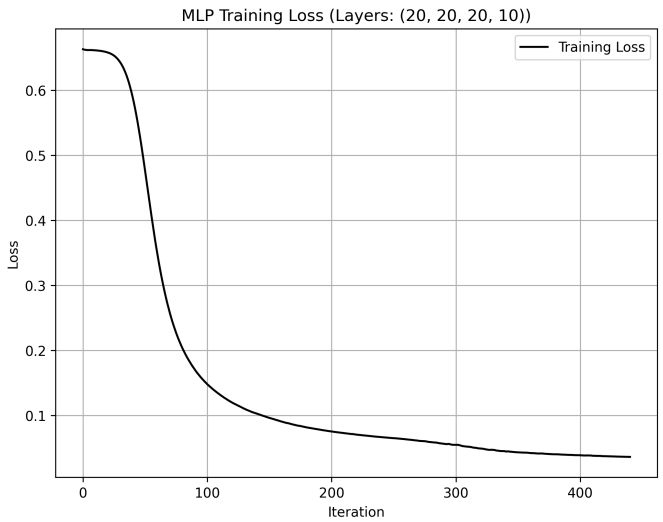

```
dataset = {
  "x": load_dataset("cancer")[0],
  "y": load_dataset("cancer")[1]
}
for mlp in mlp_of:
    mlp_of[mlp].setDataset(
      dataset["x"], dataset["y"])
    mlp_of[mlp].train()
    print(f"Accuracy for {mlp} layers:"+
      f" {mlp_of[mlp].get_accuracy()}")
    mlp_of[mlp].plot_training_loss(cmap=cmpl[mlp])
    mlp_of[mlp].plot_confusion_matrix(cmap=cmpl[mlp])
    print("-"*40, end="\n\n")
```

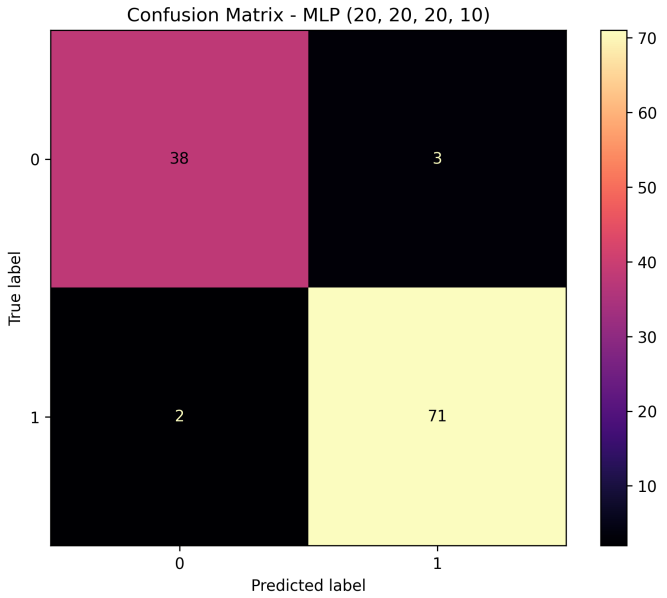
```
----------------------------------------

Accuracy for 4 layers: 0.956140350877193
```

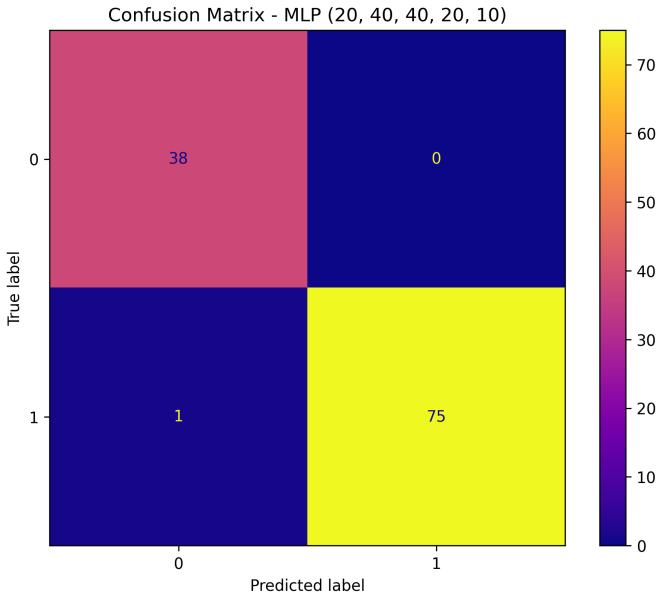
2) Cancer Dataset:

```
Accuracy for 3 layers: 0.9912280701754386
```



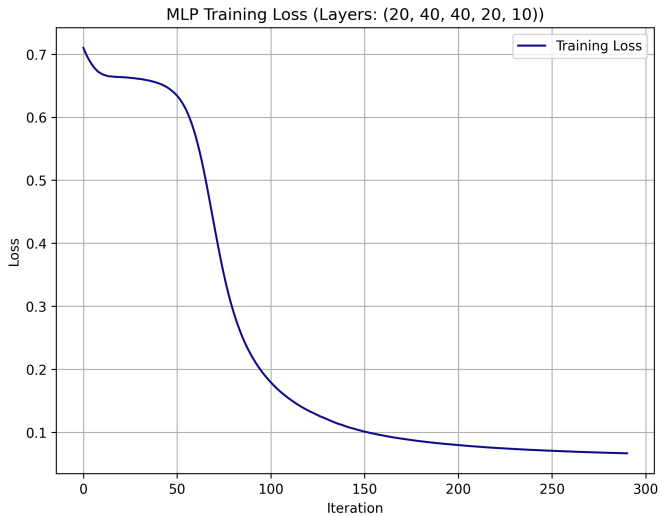


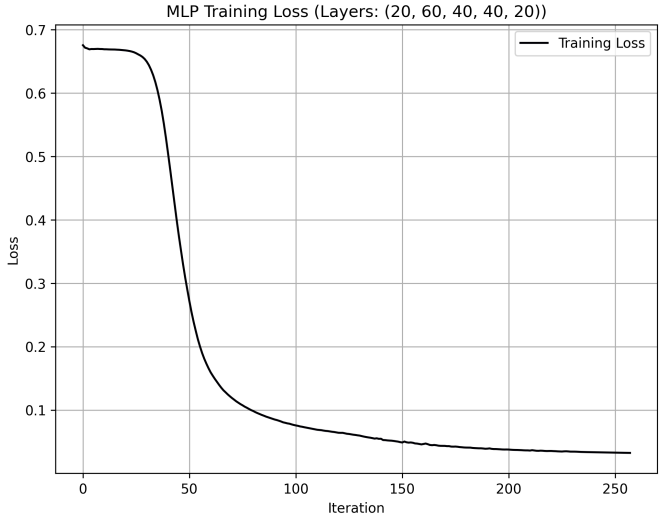


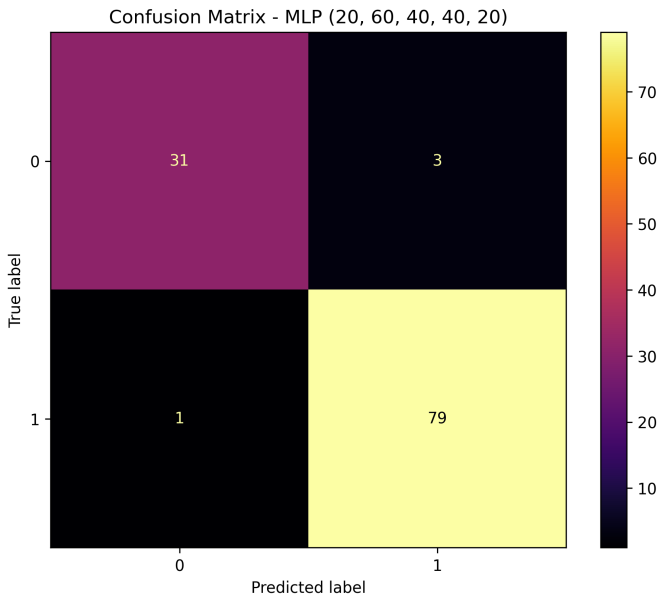


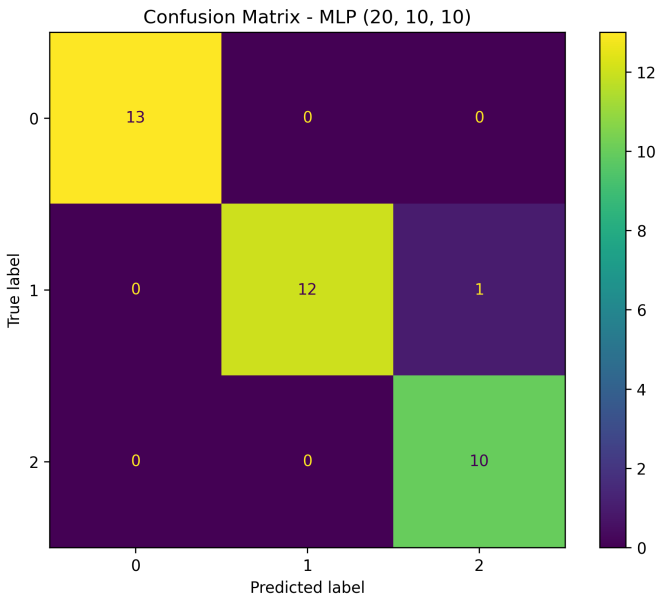
Accuracy for 5 layers: 0.9912280701754386

Accuracy for 6 layers: 0.9649122807017544







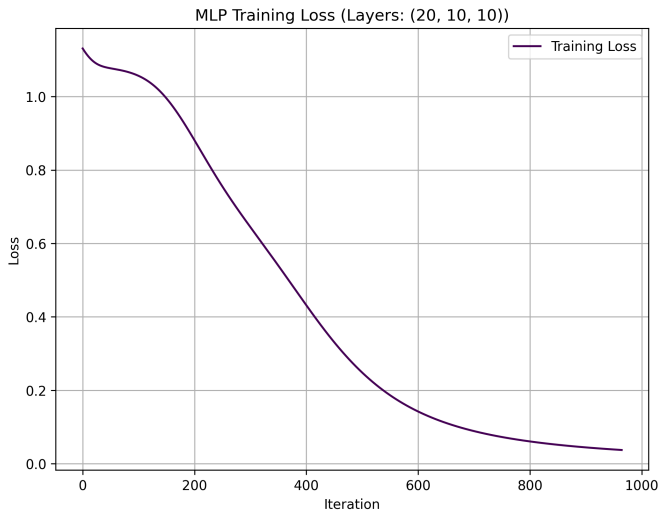


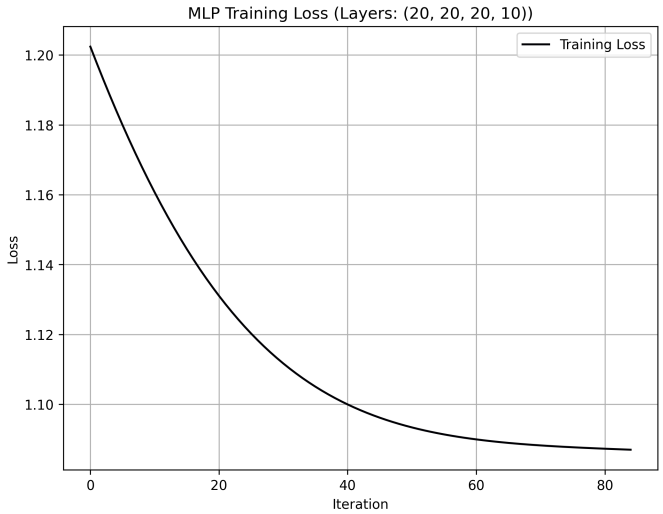

```
dataset = {
  "x": load_dataset("wine")[0],
  "y": load_dataset("wine")[1]
}
for mlp in mlp_of:
    mlp_of[mlp].setDataset(
      dataset["x"], dataset["y"])
    mlp_of[mlp].train()
    print(f"Accuracy for {mlp} layers:"+
      f" {mlp_of[mlp].get_accuracy()}")
    mlp_of[mlp].plot_training_loss(cmap=cmpl[mlp])
    mlp_of[mlp].plot_confusion_matrix(cmap=cmpl[mlp])
    print("-"*40, end="\n\n")
```

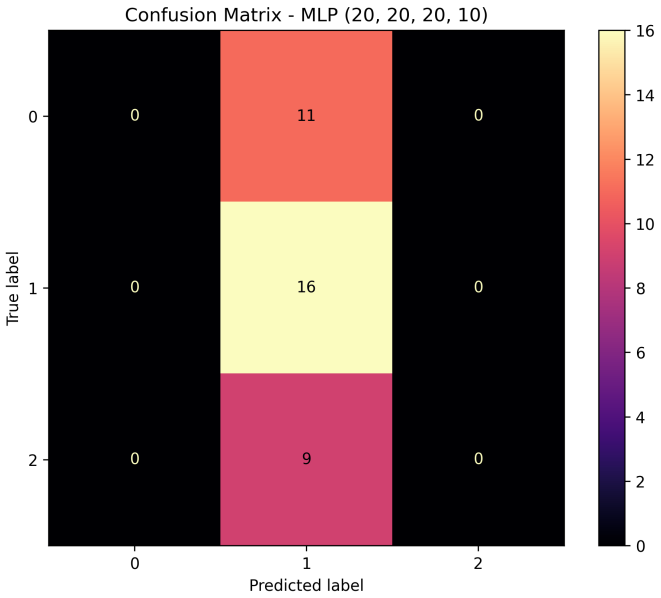
```
----------------------------------------

Accuracy for 4 layers: 0.4444444444444444
```

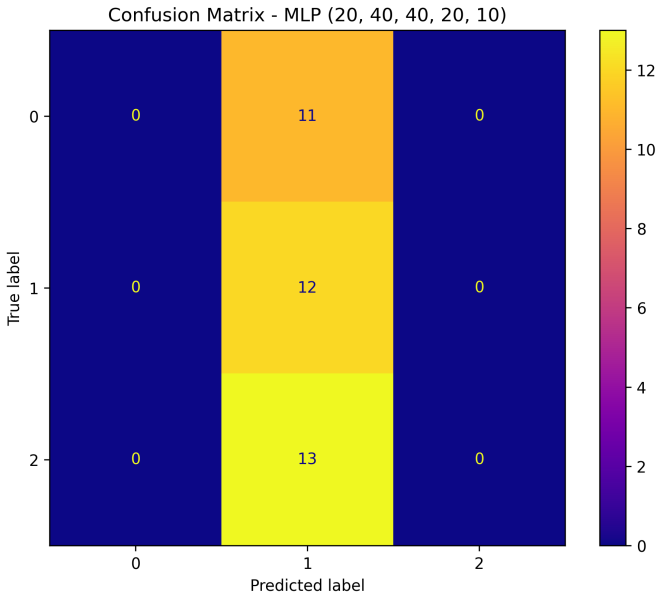
3) Wine Dataset:

```
Accuracy for 3 layers: 0.9722222222222222
```



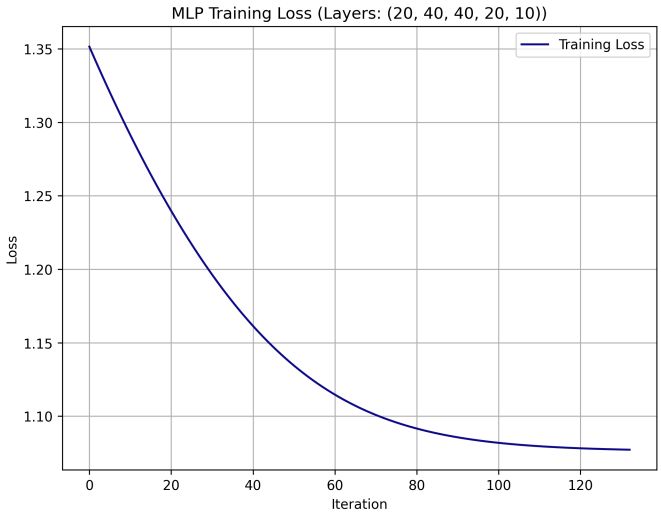


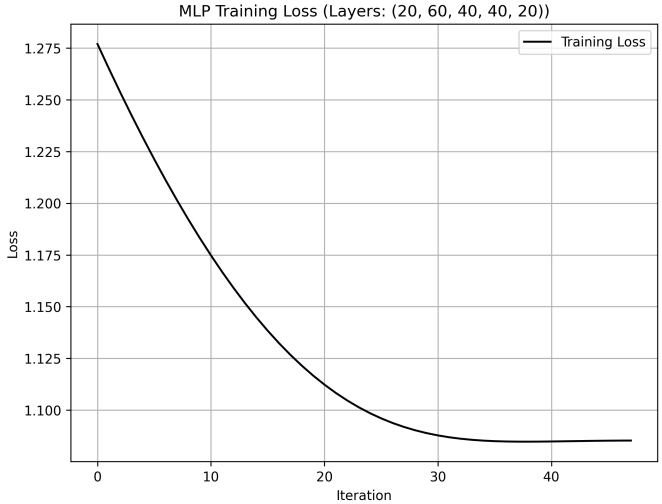


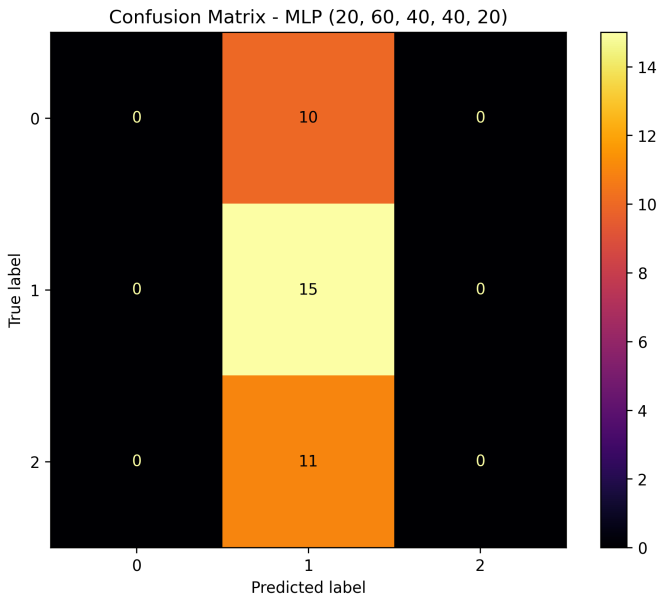


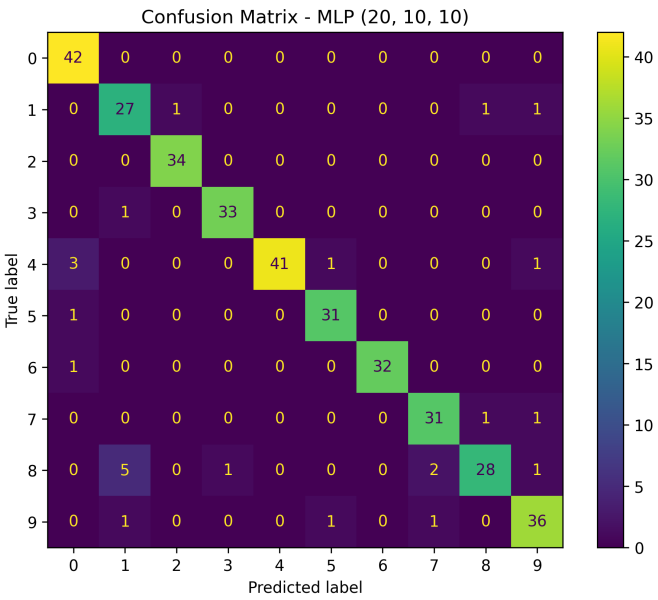
Accuracy for 5 layers: 0.3333333333333333

Accuracy for 6 layers: 0.4166666666666667







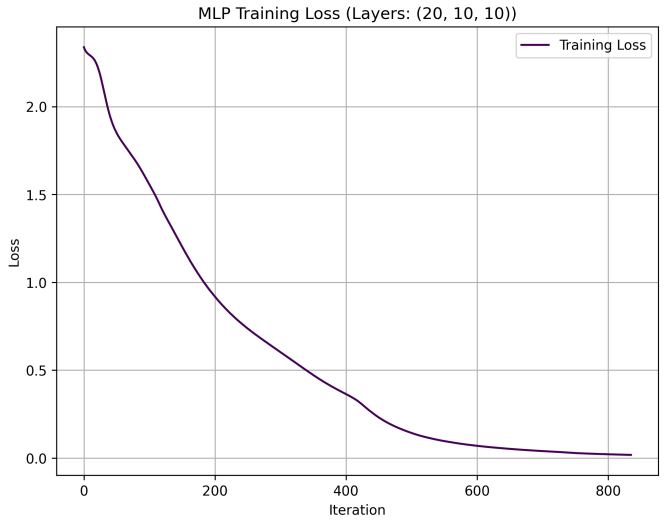


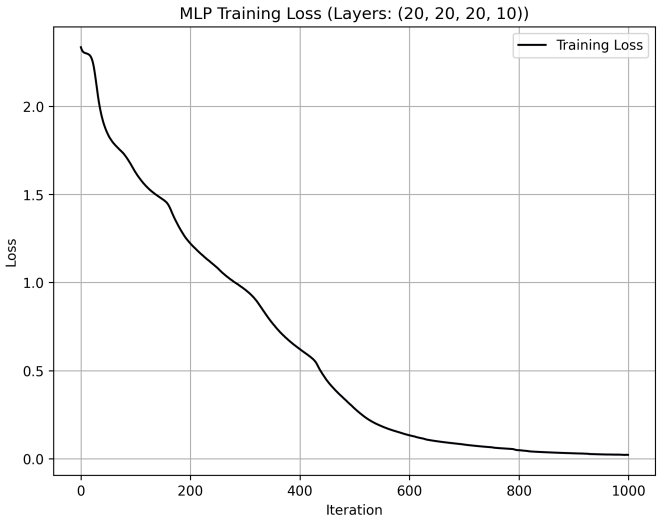

```
dataset = {
  "x": load_dataset("digits")[0],
  "y": load_dataset("digits")[1]
}
for mlp in mlp_of:
    mlp_of[mlp].setDataset(
      dataset["x"], dataset["y"])
    mlp_of[mlp].train()
    print(f"Accuracy for {mlp} layers:"+
      f" {mlp_of[mlp].get_accuracy()}")
    mlp_of[mlp].plot_training_loss(cmap=cmpl[mlp])
    mlp_of[mlp].plot_confusion_matrix(cmap=cmpl[mlp])
    print("-"*40, end="\n\n")
```

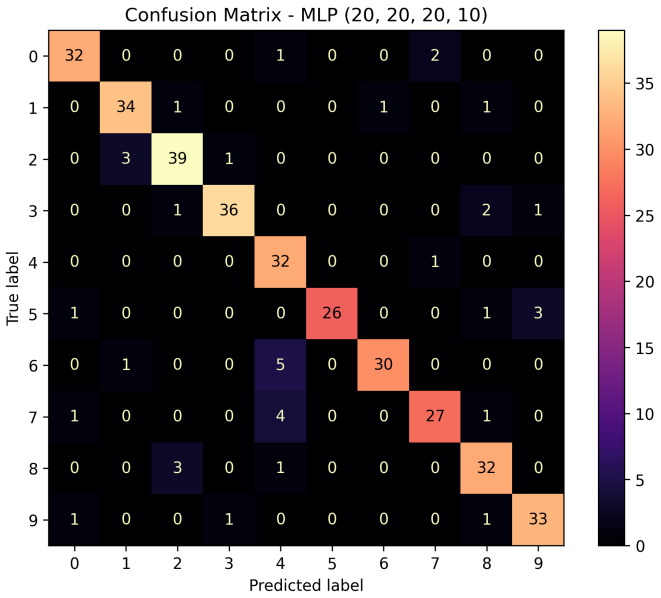
```
----------------------------------------
Accuracy for 4 layers: 0.8916666666666667
```

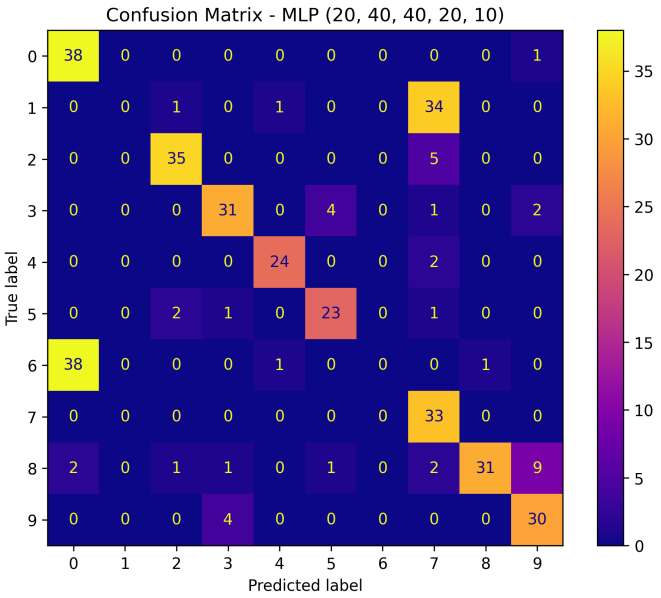
4) Digits Dataset:

```
Accuracy for 3 layers: 0.9305555555555556
```



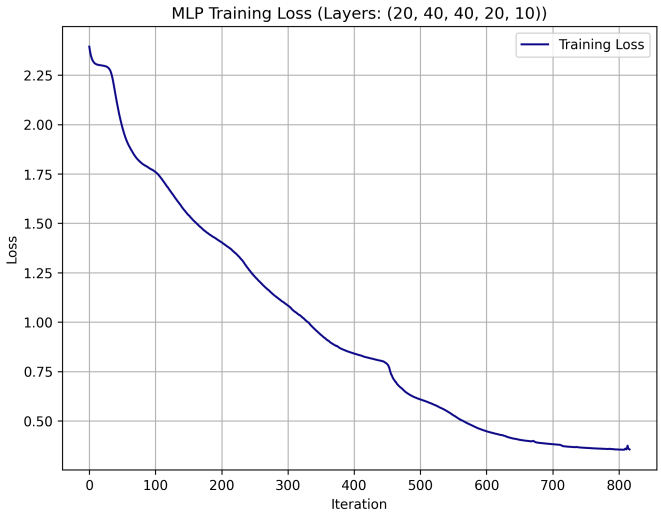


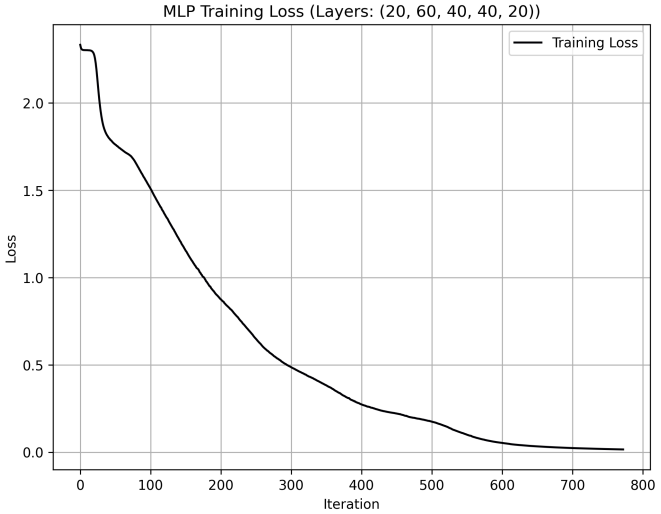


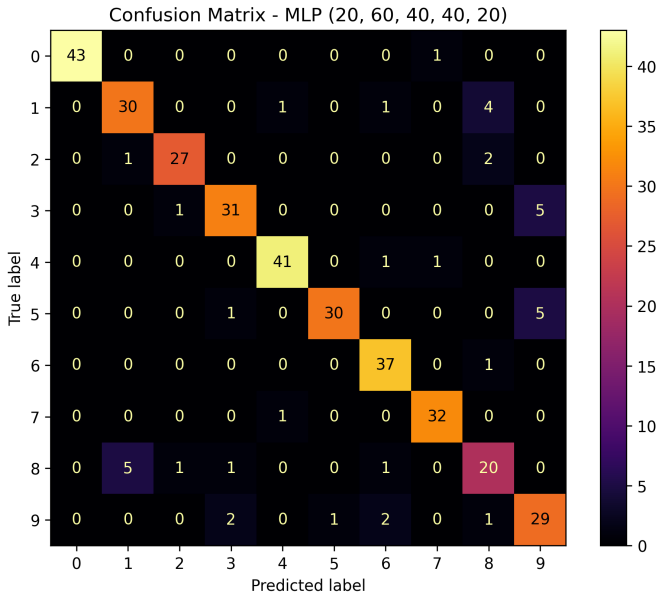


Accuracy for 5 layers: 0.6805555555555556

Accuracy for 6 layers: 0.8888888888888888







V. Conclusions

As seen above, multilayer perceptrons has been able to classify almost all the datasets. Depending on the perceptron size, and per layer number of neurons, the accuracy will be different. In some cases, the performance could be better, in other cases, as in the 4 layers one, it could be worse.

References

[1] M. Ekman. *Learning Deep Learning: Theory and Practice of Neural Networks, Computer Vision, Natural Language Processing, and Transformers Using TensorFlow*. Pearson Education, 2021. ISBN: 9780137470297. URL: https://books.google.com.mx/books?id=wNnPEAAAQBAJ.

[2] S.S. Haykin. *Neural Networks and Learning Machines*. Neural networks and learning machines v. 10. Prentice Hall, 2009. ISBN: 9780131471399. URL: https://books.google.com.mx/books?id=K7P36lKzI_QC.

[3] quarto. *Guide*. URL: https://quarto.org/docs/guide/.

[4] quarto. *PDF Options*. URL: https://quarto.org/docs/reference/formats/pdf.html.

[5] quarto. *Welcome Page*. URL: https://quarto.org.

[6] scikit-learn. *MLPClassifier*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html.

[7] Parul Singh. *A Must-Read History of Artificial Intelligence*. 2020. URL: https://cyfuture.com/blog/history-of-artificial-intelligence/.