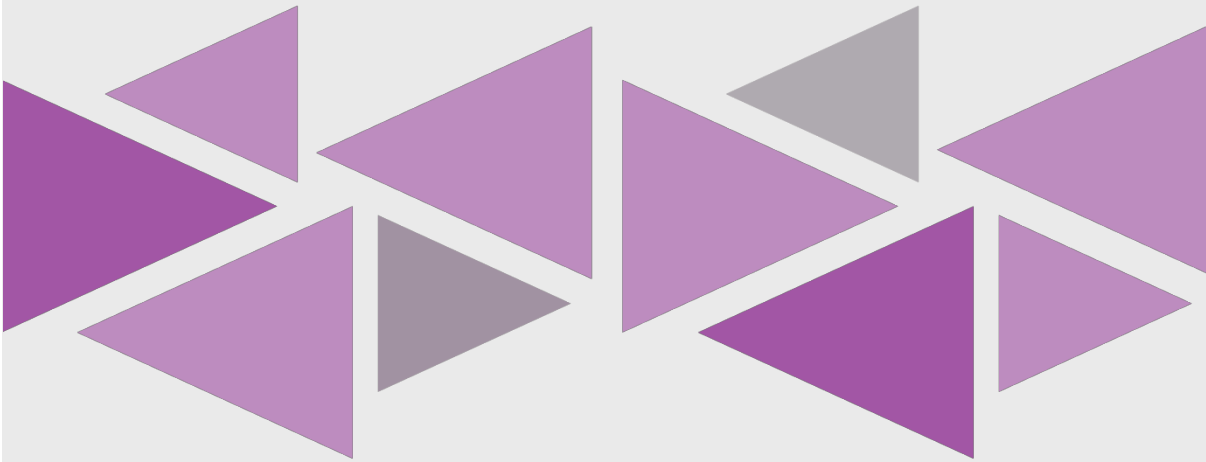


2024



MUSHROOM EDIBILITY PREDICTION PROJECT REPORT



Eren KIZILIRMAK

Tunahan Türker ERTÜRK

Berru HANEDAR

210704025

200706019

200706028

CONTENTS

ABSTRACT	3
1. INTRODUCTION.....	4
1.1 Project Overview	4
1.2 Literature Review	4
2. ABOUT THE PROJECT	7
2.1 Dataset	7
3. METHODOLOGY.....	9
3.1. Data Preprocessing	9
3.1.1. Exploratory Data Analysis (EDA)	9
3.1.2. Creating Dataset Object	11
3.1.3. Creating DataLoader Object	11
3.2. Model Architecture Design	12
3.3. Model Implementation.....	12
3.3.1. Layer Structure.....	13
3.3.2. Layer Components.....	13
3.4. Model Training & Validation.....	14
3.4.1 Training Components	14
3.4.2. Performance Monitoring on Each Epoch.....	15
3.5. Model Testing.....	16
3.5.1. Performance Metrics	16
3.5.2. Confusion Matrix Visualization.....	17
4. OBSERVATIONS	18
4.1 Errors Encountered.....	18
5. CONCLUSIONS	19
6. APPENDIX	20
6.1 Imports	20
6.2 Data Preprocessing	20
6.3. Model Implementation	22

T.C. MALTEPE UNIVERSITY
FACULTY OF ENGINEERING AND NATURAL SCIENCES
CEN 451 01: ARTIFICIAL NEURAL NETWORKS

6.4. Training Model	23
6.5. Model Testing	25
6.6. Visualizations	26
7. REFERENCES	28

ABSTRACT

This report summarizes CEN 451 01: Artificial Neural Networks' course project about Mushroom Edibility Prediction, which we gained experience in ML project development and testing. In addition to project work, we also used this opportunity to develop programming skills, learning Pytorch and MLOPs. This report consists of an overview of our experience, including the skills and knowledge we gained, the challenges we faced during the development time of this project. At the end, we give a brief conclusion about the project and its future effects on our career.

1. INTRODUCTION

The main goal of this project is to gain experience in MLPs. We did that by learning to design and implement a 3-layered MLP Neural Network. It was also important to improve our programming skills in Python. Besides all the technical work we have also learned to work in a team and learned to divide tasks among on team members. This introduction is an overview of our activities and a brief literature survey on the technical topics. Here is what we did and some background information on the main topics we studied.

1.1 Project Overview

Throughout development of the project, We learned Pytorch for building the core ML blocks, Matplotlib & Seaborn for plotting and monitoring, Scikit-learn for applying different metrics, Numpy for array operations and Pandas for statistical exploratory data analysis.

We firstly selected a dataset to build our model on, in our case Mushroom dataset. Then, we designed an abstract prediction model. Then, we learned to follow our diagram by using Pytorch.

Towards the end part of the project, We discovered more advanced topics such as Model Monitoring, Batch Normalization for better performance and applying applying ADAM optimizer for momentum and dynamic learning rate control.

1.2 Literature Review

Python Programming Language

Python is a high-level, interpreted programming language introduced by Guido van Rossum in 1991. Python is known for its readability, ease of use, and extensive ecosystem of Libraries, making it a popular choice for data science, machine learning, and web development [1]. Python's type safety is complemented by its dynamic Typing and Strong Typing, which help reduce runtime errors and enhance developer productivity.

Git

Git, a distributed version control system developed by Linus Torvalds in 2005, is designed to handle projects of any size with speed and efficiency. Chacon & Straub detail how Git enables multiple developers to work on the same project simultaneously, tracking changes and managing different code versions [2].

Github

GitHub, a web based hosting service for Git repositories, extends Git's capabilities. Bell & Beer (2014) discuss how GitHub is a platform for collaboration, code sharing, and version control, offering features like pull requests, issue tracking, and project management tools [3].

VS Code IDE

Visual Studio Code (VS Code) is a Source Code Editor developed by Microsoft. It supports a wide range of programming languages and includes features such as Integrated Terminal, Debugging Support, and Extensions for Additional Functionality. VS Code is highly extensible and can be customized to fit various development needs, making it a popular choice for developers [4].

Pytorch

PyTorch is an open-source machine learning library developed by Facebook's AI Research lab. It is known for its Flexibility and Dynamic Computational Graph, which makes it suitable for a wide range of machine learning tasks, including building and training neural networks. PyTorch provides a seamless transition from research to production, making it a preferred choice for both academia and industry [5].

Scikit-learn

Scikit-learn is a Python library for machine learning that provides simple and efficient tools for data mining and data analysis. It is built on NumPy, SciPy, and Matplotlib and is designed to be accessible and reusable in various contexts. Scikit-learn includes a wide range of algorithms for classification, regression, clustering, and dimensionality reduction, making it a valuable tool for machine learning practitioners [6].

Numpy

NumPy is a fundamental package for scientific computing with Python. It provides support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. NumPy is essential for data manipulation and is widely used in data science and machine learning [7].

Pandas

Pandas is a powerful data manipulation and analysis library for Python. It provides data structures and operations for manipulating numerical tables and time series. Pandas is built on top of NumPy and is designed to handle real-world data, making it a crucial tool for data preprocessing and analysis [8].

MLP

A Multi-Layer Perceptron (MLP) is a class of feedforward artificial neural networks. MLPs are composed of multiple layers of nodes in a directed graph, with each layer fully connected to the next one. They are widely used for various tasks, including binary classification, due to their ability to model non-linear relationships between input features and output labels. MLPs are particularly effective in high-dimensional spaces and can be trained using backpropagation and gradient descent algorithms [9].

Matplotlib

Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK. Matplotlib is highly customizable and is widely used for creating static, animated, and interactive visualizations in Python [10].

Seaborn

Seaborn is a data visualization library based on Matplotlib. It provides a high-level interface for drawing attractive statistical graphics. Seaborn simplifies the creation of complex visualizations, such as heatmaps, timeSeries plots, and distribution plots, making it a valuable tool for data exploration and presentation [11].

2. ABOUT THE PROJECT

This project focuses on developing a neural network model to classify mushrooms as edible or poisonous based on their physical characteristics. The implementation uses the UCI Mushroom Dataset, which gives a collection of mushroom samples with different attributes that can be used to predict edibility.

2.1 Dataset

The UCI Mushroom Dataset, created by Jeff Schlimmer in 1981, consists of 8,124 mushroom samples from 23 species of gilled mushrooms in the Agaricus and Lepiota families. Each mushroom in the dataset is labeled as edible or poisonous based on field guides. The dataset provides 23 different categorical attributes for each mushroom sample.

Table 1. Dataset attributes description with their values

Attributes	Values & Description
Class	edible=e, poisonous=p
Cap-shape	bell=b, conical=c, convex=x, flat=f, knobbed=k, sunken=s
Cap-surface	fibrous=f, grooves=g, scaly=y, smooth=s
Cap-color	brown=n, buff=b, cinnamon=c, gray=g, green=r, pink=p, purple=u, red=e, white=w, yellow=y
Bruises	bruises=t, no=f
Odor	almond=a, anise=l, creosote=c, fishy=y, foul=f, musty=m, none=n, pungent=p, spicy=s
Gill-attachment	attached=a, descending=d, free=f, notched=n
Gill-spacing	close=c, crowded=w, distant=d
Gill-size	broad=b, narrow=n

T.C. MALTEPE UNIVERSITY
FACULTY OF ENGINEERING AND NATURAL SCIENCES
CEN 451 01: ARTIFICIAL NEURAL NETWORKS

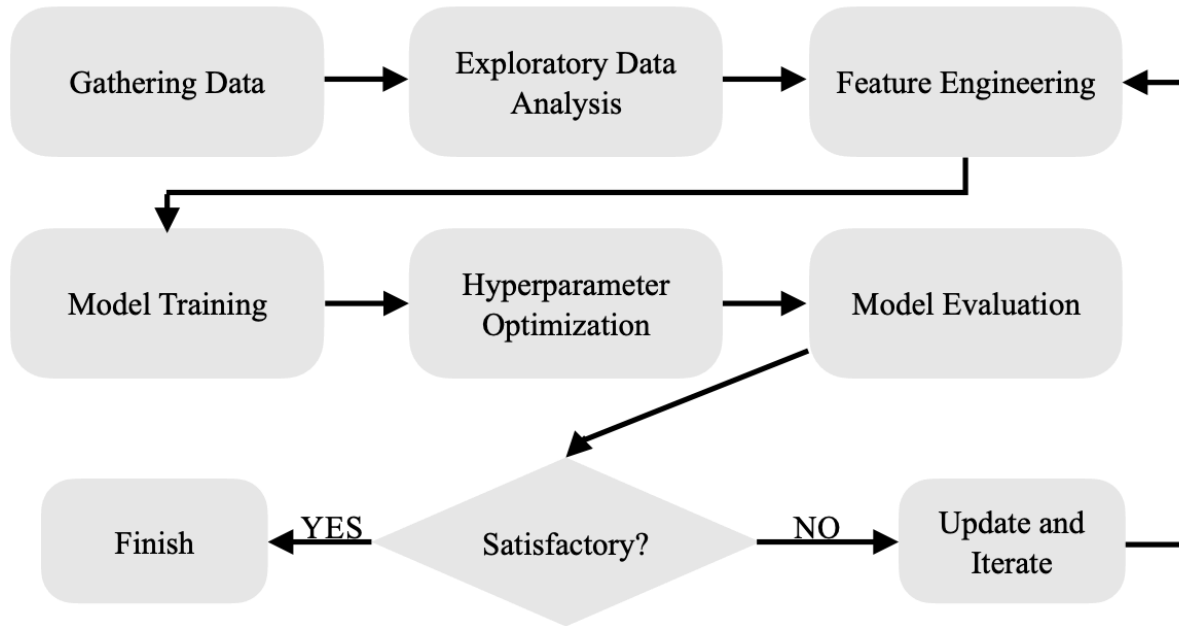
Gill-color	black=k,brown=n,buff=b,chocolate=h,gray=g, green=r,orange=o,pink=p,purple=u,red=e,white=w,yellow=y
Stalk-shape	enlarging=e,tapering=t
Stalk-root	bulbous=b,club=c,cup=u,equal=e,rhizomorphs=z,rooted=r,missing=?
Stalk-surface-above-ring	fibrous=f,scaly=y,silky=k,smooth=s
Stalk-surface-below-ring	fibrous=f,scaly=y,silky=k,smooth=s
Stalk-color-above-ring	brown=n,buff=b,cinnamon=c,gray=g,orange=o,pink=p,red=e,white=w,yellow=y
Stalk-color-below-ring	brown=n,buff=b,cinnamon=c,gray=g,orange=o,pink=p,red=e,white=w,yellow=y
Veil-type	partial=p,universal=u
Veil-color	brown=n,orange=o,white=w,yellow=y
Ring-number	none=n,one=o,two=t
Ring-type	cobwebby=c,evanescent=e,flaring=f,large=l, none=n,pendant=p,sheathing=s,zone=z
Spore-print-color	black=k,brown=n,buff=b,chocolate=h,green=r,orange=o,purple=u,white=w,yellow=y
Population	abundant=a,clustered=c,numerous=n,scattered=s,several=v,solitary=y
Habitat	grasses=g,leaves=l,meadows=m,paths=p,urban=u,waste=w,woods=d

The dataset has no missing values and consists only of categorical attributes. For neural network, preprocessing steps including one-hot encoding are necessary to convert the categorical data into numerical format (float32). The dataset contains 4,208 edible samples and 3,916 poisonous samples a balanced distribution of the target classes. This is important because it will not cause problems such as bias.

3. METHODOLOGY

Methodology for this project is shown briefly in the diagram below the section;

Figure 1. Methodology Diagram



3.1. Data Preprocessing

3.1.1. Exploratory Data Analysis (EDA)

We applied EDA technique for the dataset. This includes using plotting to check distributions of the columns, correlations, checking missing data and unique values for each columns. Distribution of the dataset was: 4,208 edible (51.8%) and 3,916 poisonous (48.2%) samples.

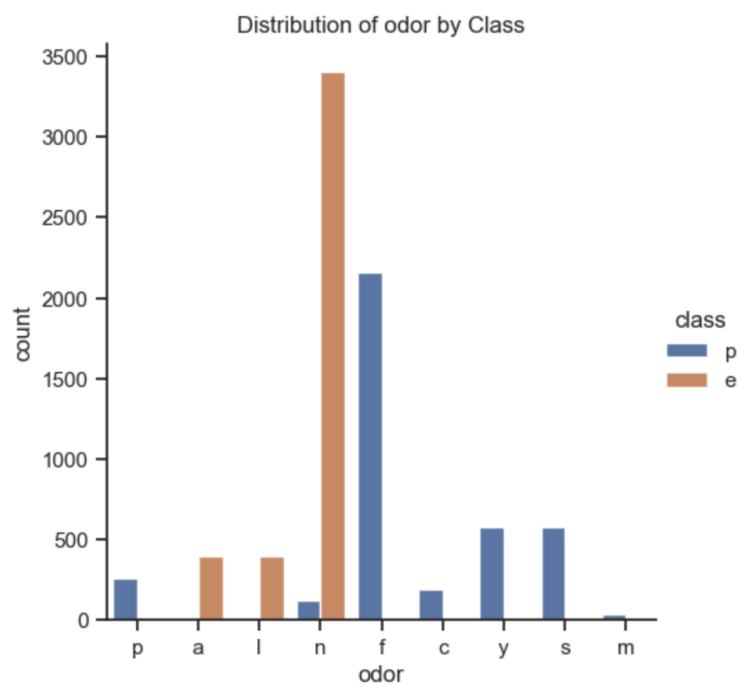
Even though no missing values were present in the dataset, Figure 2. shows there was a column, veil-type, which had only 1 unique value for the entire dataset. We dropped that column because it hadn't provided any useful information.

Figure 2. Unique values for each column



Odor emerged as a potentially strong predictor, with certain odors (e.g., foul, pungent) strongly correlated with poisonous mushrooms. Figure 3. shows the distribution of odor and it's relation to poisonous class.

Figure 3. Distribution of Odor by Class



3.1.2. Creating Dataset Object

We created a custom Dataset class inheriting from PyTorch's Dataset class to handle our mushroom data. Our implementation targeted Apple Silicon hardware by using the MPS (Metal Performance Shaders) device for better computation. We created 3 sets of data using the dataset. first one, training, is only used for both forward propagation and back propagation. It's the 70% of the dataset. second one, validation, is only used during training to calculate the validation accuracy and validation loss to make model robust to unseen data. It's the 15% of the data. Third one, test, is created for only testing which is important to test the model's final robustness. It's the 15% of the data.

Figure 4. Dataset Object Code

```
class dataset(Dataset):
    def __init__(self,X, Y):
        self.X = torch.tensor(X, dtype= torch.float32).to(device="mps")
        self.Y = torch.tensor(Y, dtype= torch.float32 ).to(device="mps")

    def __len__(self):
        return len(self.X)

    def __getitem__(self, index):
        return self.X[index], self.Y[index]

train_data = dataset(X_train,Y_train)
test_data = dataset(X_test,Y_test)
validation_data = dataset(X_val,Y_val)
# from torch.utils.data import TensorDataset

#blabla_data = TensorDataset(torch.tensor(X, dtype= torch.float32), torch.tensor(Y, dtype= torch.float32) )
train_data.__getitem__(2)

tensor([0., 0., 0., 0., 0., 1., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0., 0., 0.,
        0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 1., 1., 0., 0.,
        1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 0., 0.,
        0., 0., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 1.,
        0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 1., 0., 0., 1., 0., 1.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 1.,
        0., 1., 0., 0., 0., 0., 0., 0.], device='mps:0'),
tensor(1., device='mps:0'))
```

3.1.3. Creating DataLoader Object

For data loading, we created a custom loader function that creates DataLoader objects with sampling configurations. DataLoader Object takes batches of samples from the given datasets. It is main purpose is to give data to the model. Figure 5. shows the code.

Figure 5. DataLoader Object Code

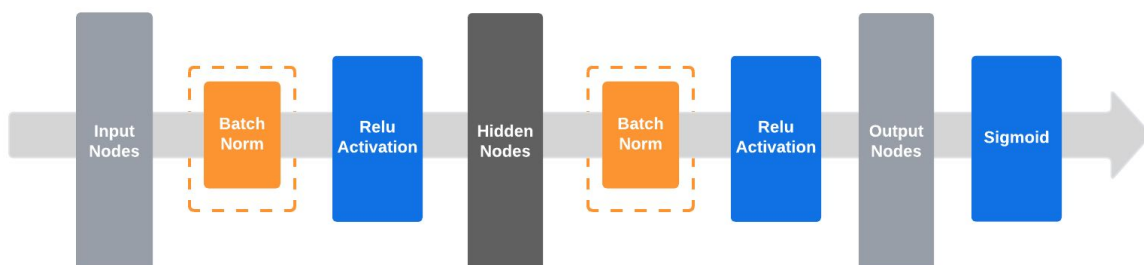
```
def loaders(train, test, val) -> dict[str,any]:  
  
    loaders = {"train":train, "test":test, "validation":val}  
  
    for key, val in loaders.items():  
  
        loaders[key] = DataLoader(dataset=val, batch_sampler =  
                                BatchSampler(  
                                    sampler=RandomSampler(val),  
                                    batch_size=64,  
                                    drop_last=True  
                                )  
        )  
  
    print(loaders)  
    return loaders["train"], loaders["test"], loaders["validation"]
```

The loaders function returns the data loaders for each dataset. It uses BatchSampler for faster calculation and it uses RandomSampler to prevent order dependent batches. It Drops last incomplete batch to ensure consistent batch sizes during training. A batch of 64 data samples is getting fed to the model for each dataset.

3.2. Model Architecture Design

The Mushroom dataset contains structured data (categorical features). MLPs are suited for such data after appropriate preprocessing, such as one-hot encoding for categorical variables.

Figure 6. Model Architecture



3.3. Model Implementation

Our neural network implementation is a multi-layer perceptron (MLP) architecture in PyTorch. The model is designed to process the mushroom features and output binary

classification predictions. Since MLPs are efficient and easier to implement, compared to more complex networks such as CNNs, which is focused for spatial features as images, MLPs were a more practical choice.

3.3.1. Layer Structure

The input layer gets the one-hot encoded features (with dimensions equal to `X.shape[1]`). The hidden layer comprises 10 neurons. The output layer consists of a single neuron with sigmoid activation, which is suitable for binary classification.

3.3.2. Layer Components

Each layer uses Linear Transformations which is implemented using `nn.Linear`. Hidden layers and Input layer Uses `BatchNorm` to prevent the incoariate shift and a `ReLU` activation to prevent vanishing gradients for faster calculation and non-linearity. Output layer only consists of a linear transformation nodes and a sigmoid Activation which is applied finally to predict the given output. Since it's a binary classification problem, using sigmoid function is suitable for scaling the output for probability. Figure 6. shows the code for the model.

Figure 7. MLP model's code

```
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()

        self.input_layer = nn.Linear(X.shape[1], 10)
        self.norm1 = nn.BatchNorm1d(10)
        self.relu1 = nn.ReLU()

        self.linear1 = nn.Linear(10, 10)
        self.norm2 = nn.BatchNorm1d(10)
        self.relu2 = nn.ReLU()

        self.linear2 = nn.Linear(10, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.input_layer(x)
        x = self.norm1(x)
        x = self.relu1(x)
        x = self.linear1(x)
        x = self.norm2(x)
        x = self.relu2(x)
        x = self.linear2(x)
        x = self.sigmoid(x)
        return x

model = MyModel()
model.to("mps")
```

Forward pass function implements how data flows through the network. Since we are creating an MLP, there is a casual forward pass without skipping layers.

3.4. Model Training & Validation

The training process was feeding our preprocessed data through the network in 64 batches. During each training iteration, The input data passes through the network, with batch normalization applied between layers. The ReLU activation function processed the normalized outputs. The Binary Cross-Entropy loss is calculated between the model's predictions and the true labels. And Adam optimizer updated the model parameters based on the computed gradients. This process was repeated for 15 epochs until the model achieved satisfactory performance on both training and validation sets.

3.4.1 Training Components

During the training, each batch of data passes through the network in a forward pass, at the end Binary Cross-Entropy loss is calculated between predictions and true labels. The model then performs back propagation and updates parameters using the Adam optimizer. This process repeats for each batch in epoch.

Figure 8. Training loop

```
epochs = 15
for epoch in range(epochs):
    total_acc_train = 0
    total_acc_val = 0
    total_loss_train = 0
    total_loss_val = 0

    for data in train_dataloader:

        #####
        #this part is training the model with batches.

        inputs, labels = data

        prediction = model(inputs).reshape(len(labels)) #we need this beacuse prediction is 8,1 dimension bu

        #print(prediction)

        batch_loss = criterion(prediction, labels)

        total_loss_train += batch_loss.item()

        acc = (prediction.round() == labels).sum().item() #calculating accuracy of a batch item !!!

        total_acc_train += acc

        batch_loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        #####

    #this part is for validation during training.
    with torch.no_grad():
        for data in validation_dataloader:
            inputs, labels = data
            prediction = model(inputs).reshape(len(labels))
            batch_loss = criterion(prediction, labels)
            total_loss_val += batch_loss.item()

            acc = (prediction.round() == labels).sum().item() |
            total_acc_val += acc
```

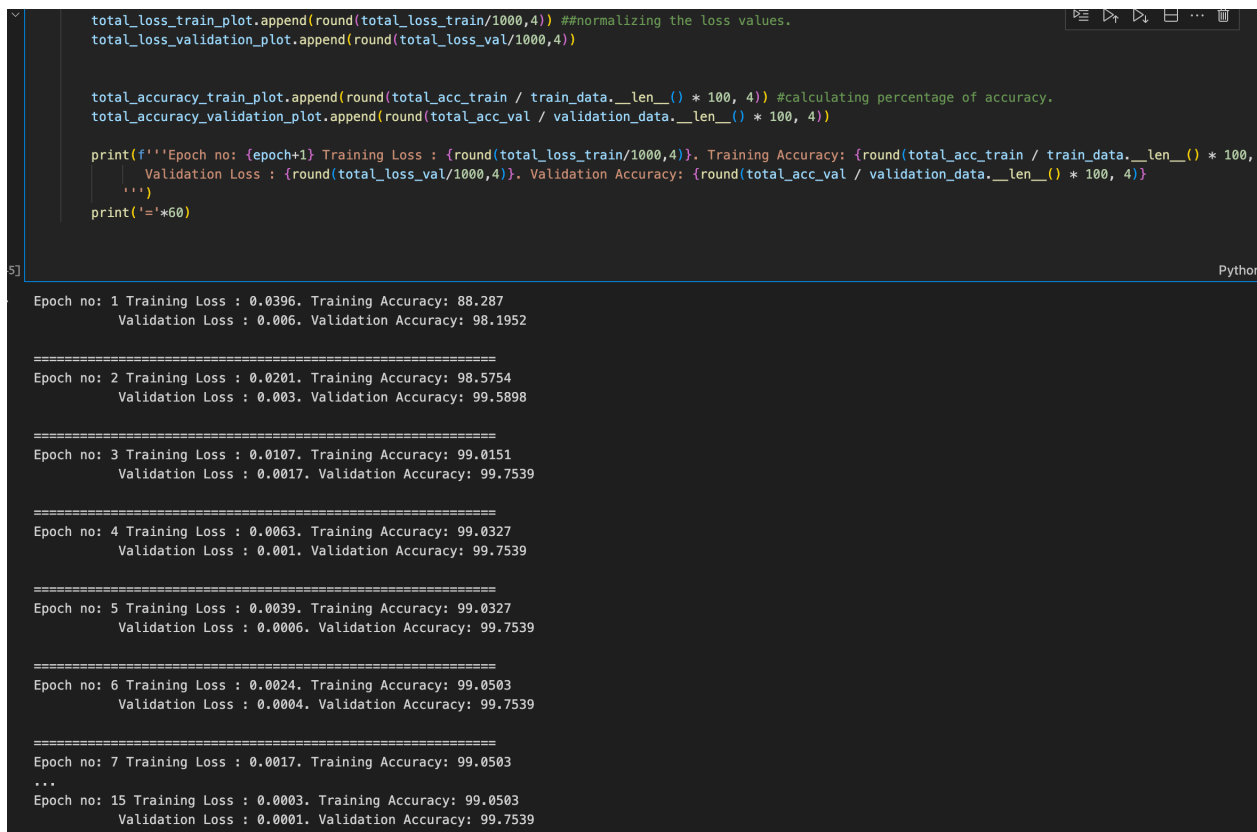
The validation phase checks the model's performance on unseen data without computing gradients. This is for monitoring the model's robustness capability and to detect potential overfitting. During validation, we compute the same metrics as training but don't apply back propagation.

3.4.2. Performance Monitoring on Each Epoch

Figure 8. tracks loss and accuracy performance metrics throughout training. The training and validation losses measure the model's prediction error on their datasets, while accuracies show the percentage of correct predictions. These metrics are normalized and stored for visualization. After each epoch, the system prints a progress report showing both training and validation performance of the model's learning progress.

The metrics has multiple roles. By maintaining different training and validation metrics, we can see how the model generalizes to unseen data and make decisions about when to stop training or adjust its parameters.

Figure 9. Monitoring Code



```
total_loss_train_plot.append(round(total_loss_train/1000,4)) ##normalizing the loss values.
total_loss_validation_plot.append(round(total_loss_val/1000,4))

total_accuracy_train_plot.append(round(total_acc_train / train_data.__len__() * 100, 4)) #calculating percentage of accuracy.
total_accuracy_validation_plot.append(round(total_acc_val / validation_data.__len__() * 100, 4))

print(f'''Epoch no: {epoch+1} Training Loss : {round(total_loss_train/1000,4)}. Training Accuracy: {round(total_acc_train / train_data.__len__() * 100,
    Validation Loss : {round(total_loss_val/1000,4)}. Validation Accuracy: {round(total_acc_val / validation_data.__len__() * 100, 4)}
    ''')
print('='*60)
```

Epoch no: 1 Training Loss : 0.0396. Training Accuracy: 88.287
Validation Loss : 0.006. Validation Accuracy: 98.1952

=====

Epoch no: 2 Training Loss : 0.0201. Training Accuracy: 98.5754
Validation Loss : 0.003. Validation Accuracy: 99.5898

=====

Epoch no: 3 Training Loss : 0.0107. Training Accuracy: 99.0151
Validation Loss : 0.0017. Validation Accuracy: 99.7539

=====

Epoch no: 4 Training Loss : 0.0063. Training Accuracy: 99.0327
Validation Loss : 0.001. Validation Accuracy: 99.7539

=====

Epoch no: 5 Training Loss : 0.0039. Training Accuracy: 99.0327
Validation Loss : 0.0006. Validation Accuracy: 99.7539

=====

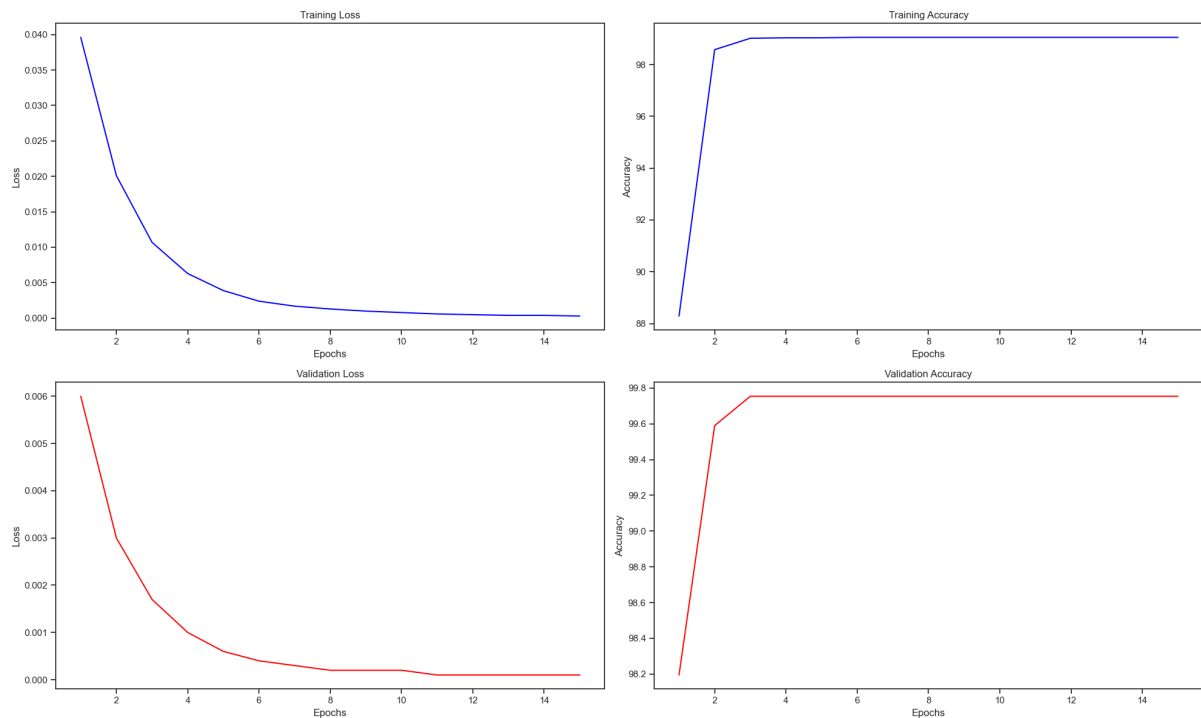
Epoch no: 6 Training Loss : 0.0024. Training Accuracy: 99.0503
Validation Loss : 0.0004. Validation Accuracy: 99.7539

=====

Epoch no: 7 Training Loss : 0.0017. Training Accuracy: 99.0503
...

Epoch no: 15 Training Loss : 0.0003. Training Accuracy: 99.0503
Validation Loss : 0.0001. Validation Accuracy: 99.7539

Figure 10. Visualization of Training and Validation per Epoch



3.5. Model Testing

After completing the training process, we tested our model's performance on the unseen test dataset to assess its generalization capability and predictive accuracy. Test implementation is the same as validation.

3.5.1. Performance Metrics

The model achieved performance on the test dataset with the its metrics Test Loss was 0.0001 and the Test Accuracy is 99.75%. To gain deeper information about model's performance, we used `classification_report` from `sklearn.metrics`. This function gave us Precision, Recall, f1 and their averages.

Figure 11. Classification Report

```
from sklearn.metrics import classification_report

matrix = classification_report(y_true=yss_arr, y_pred=preds_arr)
print(matrix)
```

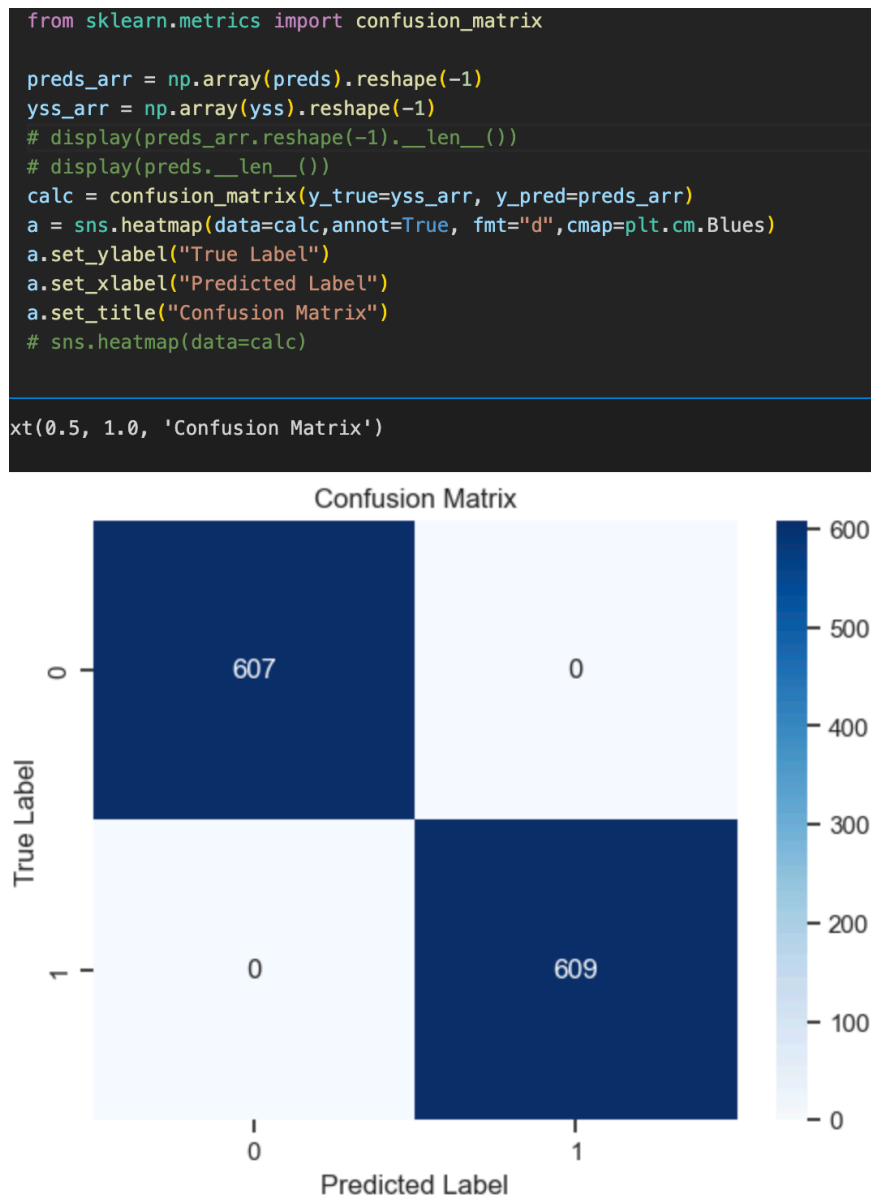
	precision	recall	f1-score	support
0.0	1.00	1.00	1.00	607
1.0	1.00	1.00	1.00	609
accuracy			1.00	1216
macro avg	1.00	1.00	1.00	1216
weighted avg	1.00	1.00	1.00	1216

3.5.2. Confusion Matrix Visualization

The confusion matrix visualization provides an aesthetic view of the model's prediction performance. The deep blue colors show strong classification performance. This shows that our model has successfully learned the eating features between edible and poisonous mushrooms from the training data and it's robust.

These results demonstrate that our neural network model has achieved robust and reliable performance in mushroom classification.

Figure 12. Confusion Matrix



4. OBSERVATIONS

During the development and implementation of our 3-layer MLP for the Mushroom dataset, there were several challenges and made observations of understanding of neural networks and their behavior. Below, we show the errors encountered, how we resolved them, and the overall results of our project.

4.1 Errors Encountered

We have encountered error which is ranged from data preprocessing to the end of the project. Very first error we encountered was incorrect handling of categorical features caused mismatches in the input dimensions of the model. We resolved this by making all categorical features one-hot encoded and matched the input size of the first layer with the dataset's feature count.

The choice of batch size also affected the training process. Smaller batch sizes led to more frequent updates of the model parameters which crashed IDE with it's increased processing time. We tried different batch sizes and found that a batch size of 64 provided a good balance between training speed and model performance.

The number of neurons in the hidden layer impacted the model's performance. Too few neurons led to underfitting, while too many neurons increased the risk of overfitting. We tried different size of nodes in each layer and found out the 10 node for each node was best.

We initially used the Tanh activation function, but it resulted in slower convergence. Switching to the ReLU activation function improved the training speed and performance.

We haven't had a issue about gradient vanishing and exploding. That's because that's likely to happen in deeper networks.

5. CONCLUSIONS

This project was a good learning experience in designing, implementing, and optimizing a Multi-Layer Perceptron (MLP) for a binary classification task using the Mushroom dataset. The gaining experience with MLPs was achieved, along with an understanding of advanced concepts in machine learning and neural network architecture.

We applied libraries such as PyTorch for model construction, Matplotlib and Seaborn for visualizations, and Scikit-learn for evaluating metrics. We have faced issues such as data preprocessing, overfitting, and runtime errors, which improved our debugging skills.

The project showed how MLPs are well suited for binary classification problems, by the final model scoring robust accuracy and generalization. Regularization techniques such as using ADAM optimizer, improved model's performance by applying momentum for gradients and dynamically improving learning rate.

Our programming skills in Python, analyzing data, using Pytorch, Scikit-learn Matplotlib, seaborn, are improved.

In conclusion, this project achieved it's technical goals and also provided us with a memorable experience and skills in artificial intelligence. It marked a milestone in our understanding of how to effectively apply neural networks to datasets.

6. APPENDIX

6.1 Imports

```
import torch
from torch import nn #This is to build the neural networks
from torch.utils.data import Dataset, DataLoader #Data loader object
import numpy as np #for array operations.
import pandas as pd
from torch.optim import Adam #This is for optimizer.
from sklearn.model_selection import train_test_split #to split the data
from sklearn.metrics import accuracy_score, roc_auc_score, roc_curve # scores
import matplotlib.pyplot as plt
import seaborn as sns #these are for viewing.
from scipy import stats #for statistics.
from torch.utils.data import BatchSampler, RandomSampler
from sklearn.metrics import confusion_matrix
device = "mps"
```

6.2 Data Preprocessing

```
mushroom = pd.read_csv("mushrooms.csv", delimiter=",")
mushroom.head(5)
mushroom.info()
print(mushroom["class"].value_counts())
a = mushroom.describe()[1:2]
a = a.transpose()
display(a) #as we can see the veil-type is irrelevant because it has only a value
a["unique"].sum()
mushroom = mushroom.drop(columns="veil-type")
Y = mushroom["class"]
```

```
mushroom_copy = mushroom.copy()
X = mushroom.drop(columns="class")
display("DATA before one-hot encoding:")
display(X)
X = pd.get_dummies(data = X, prefix=[column for column in X.columns], dtype=float)
display("DATA after one-hot encoding:")
display(X)
#one-hot encoding the target value as well
mapper = {"p" : 1.0, "e":0.0}
Y = Y.map(mapper)
display(Y)
##Train test splitting =
X = X.to_numpy()
Y = Y.to_numpy()
display(X)
display(Y)
X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size=0.3)
X_test, X_val, Y_test, Y_val = train_test_split(X_test, Y_test, test_size = 0.5)
print (X_train.shape)
print (X_val.shape)
print (X_test.shape)
class dataset(Dataset):
    def __init__(self,X, Y):
        self.X = torch.tensor(X, dtype= torch.float32).to(device="mps")
        self.Y = torch.tensor(Y, dtype= torch.float32 ).to(device="mps")
    def __len__(self):
        return len(self.X)
    def __getitem__(self, index):
        return self.X[index], self.Y[index]
```

```
train_data = dataset(X_train,Y_train)
test_data = dataset(X_test,Y_test)
validation_data = dataset(X_val,Y_val)
# from torch.utils.data import TensorDataset
#blabla_data = TensorDataset(torch.tensor(X, dtype = torch.float32), torch.tensor(Y,
dtype= torch.float32) )
train_data.__getitem__(2)
def loaders(train, test, val) -> dict[str,any]:
    loaders = {"train":train, "test":test, "validation":val}
    for key, val in loaders.items():
        loaders[key] = DataLoader(dataset=val, batch_sampler =
                                BatchSampler(
                                    sampler=RandomSampler(val),
                                    batch_size=64,
                                    drop_last=True
                                )
        )
    print(loaders)
    return loaders["train"], loaders["test"], loaders["validation"]
train_dataloader, test_dataloader, validation_dataloader = loaders(train=train_data,
test=test_data, val=validation_data)
for x, y in train_dataloader:
    print(x)
    print("=====")
    print(y)
    break
```

6.3. Model Implementation

```
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
```

```
self.input_layer = nn.Linear(X.shape[1], 10)
self.norm1 = nn.BatchNorm1d(10)
self.relu1 = nn.ReLU()
self.linear1 = nn.Linear(10, 10)
self.norm2 = nn.BatchNorm1d(10)
self.relu2 = nn.ReLU()
self.linear2 = nn.Linear(10, 1)
self.sigmoid = nn.Sigmoid()

def forward(self, x):
    x = self.input_layer(x)
    x = self.norm1(x)
    x = self.relu1(x)
    x = self.linear1(x)
    x = self.norm2(x)
    x = self.relu2(x)
    x = self.linear2(x)
    x = self.sigmoid(x)

    return x

model = MyModel()
model.to("mps")
```

6.4. Training Model

```
criterion = nn.BCELoss() #defining the loss function of my model.
optimizer = Adam(model.parameters(), lr= 1e-3) #defining adam optimizer and it's
learning rate.
total_loss_train_plot = []
total_loss_validation_plot = [] #values needed to show them in a plot for each epoch.
total_accuracy_train_plot = []
total_accuracy_validation_plot = []
```


epochs = 15

for epoch in range(epochs):

 total_acc_train = 0

 total_acc_val = 0

 total_loss_train = 0

 total_loss_val = 0

 for data in train_dataloader:

 #####

 #this part is training the model with batches.

 inputs, labels = data

 prediction = model(inputs).reshape(len(labels)) #we need this beacuse prediction is 8,1 dimension but criterion only takes 8 as dimension.

 #print(prediction)

 batch_loss = criterion(prediction, labels)

 total_loss_train += batch_loss.item()

 acc = (prediction.round() == labels).sum().item() #calculating accuracy of a batch item !!!

 total_acc_train += acc

 batch_loss.backward()

 optimizer.step()

 optimizer.zero_grad()

 #####

 #this part is for validation during training.

 with torch.no_grad():

 for data in validation_dataloader:

 inputs, labels = data

 prediction = model(inputs).reshape(len(labels))

 batch_loss = criterion(prediction, labels)

 total_loss_val += batch_loss.item()

 acc = (prediction.round() == labels).sum().item()

 total_acc_val += acc

```
total_loss_train_plot.append(round(total_loss_train/1000,4)) ##normalizing the loss values.
```

```
total_loss_validation_plot.append(round(total_loss_val/1000,4))
```

```
total_accuracy_train_plot.append(round(total_acc_train / train_data.__len__() * 100, 4)) #calculating percentage of accuracy.
```

```
total_accuracy_validation_plot.append(round(total_acc_val / validation_data.__len__() * 100, 4))
```

```
print(f"Epoch no: {epoch+1} Training Loss : {round(total_loss_train/1000,4)}. Training Accuracy: {round(total_acc_train / train_data.__len__() * 100, 4)}")
```

```
Validation Loss : {round(total_loss_val/1000,4)}. Validation Accuracy: {round(total_acc_val / validation_data.__len__() * 100, 4)}
```

```
    """)
```

```
print('='*60)
```

6.5. Model Testing

```
preds = []
```

```
yss = []
```

```
with torch.no_grad():
```

```
    total_loss_test = 0
```

```
    total_acc_test = 0
```

```
    for data in test_dataloader:
```

```
        inputs, labels = data
```

```
        prediction = model(inputs).reshape(len(labels))
```

```
        batch_loss = criterion(prediction, labels)
```

```
        total_loss_test += batch_loss.item()
```

```
        acc = (prediction.round() == labels).sum().item()
```

```
        total_acc_test += acc
```

```
        preds.append(prediction.round().flatten().tolist())
```

```
        yss.append(labels.reshape(len(labels)).flatten().tolist())
```

```
print(f'Test Loss : {round(total_loss_test/1000,4)}. Test Accuracy: {round(total_acc_test / test_data.__len__() * 100, 4)}')
```

6.6. Visualizations

```
## this code block is for visualization.
fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(20, 12))
epochs = [i for i in range(1, 16)]
# Define your data, titles, and labels
y_vals = [
    total_loss_train_plot,
    total_accuracy_train_plot,
    total_loss_validation_plot,
    total_accuracy_validation_plot
]
titles = [
    "Training Loss",
    "Training Accuracy",
    "Validation Loss",
    "Validation Accuracy"
]
y_labels = ["Loss", "Accuracy", "Loss", "Accuracy"]
# Define colors for the plots (training: default, validation: red)
colors = ["blue", "blue", "red", "red"]
# Flatten the axs array for easy iteration
axs_flat = axs.flatten()
# Loop through and plot
for i, (ax, y_val, title, y_label, color) in enumerate(zip(axs_flat, y_vals, titles, y_labels,
colors)):
    sns.lineplot(x=epochs, y=y_val, ax=ax, color=color)
    ax.set_title(title)
    ax.set_ylabel(y_label)
    ax.set_xlabel("Epochs")
plt.tight_layout()
```

```
plt.show()

for col in mushroom_copy.columns:
    if col != "class": # Assuming "class" is the target variable
        sns.catplot(data=mushroom_copy, x=col, hue="class", kind="count")
        plt.title(f"Distribution of {col} by Class")
        plt.show()

from sklearn.metrics import confusion_matrix
preds_arr = np.array(preds).reshape(-1)
yss_arr = np.array(yss).reshape(-1)
# display(preds_arr.reshape(-1).__len__())
# display(preds.__len__())
calc = confusion_matrix(y_true=yss_arr, y_pred=preds_arr)
a = sns.heatmap(data=calc,annot=True, fmt="d",cmap=plt.cm.Blues)
a.set_ylabel("True Label")
a.set_xlabel("Predicted Label")
a.set_title("Confusion Matrix")
# sns.heatmap(data=calc)

from sklearn.metrics import classification_report
matrix = classification_report(y_true=yss_arr, y_pred=preds_arr)
print(matrix)
```

7. REFERENCES

- [1] Van Rossum, G. (1991). Python 0.9.0.
- [2] Chacon, S. L., & Straub, b. (2014). Pro Git. Apress.
- [3] Bell, T., & Beer, d. (2014). thegithub alternative. palgrave macmillan.
- [4] Microsoft. (2021). Visual studio code.
- [5] Paszke w, a. (2019). pytorch: an imperative style, high-performance deep learning library.
- [6] Pedreggo, j. (2018). scikit-learn: machine learning in python.
- [7] Oliphant, w. h., & van der walt, j. (2020). numpy: the fundamental package for numerical computation in python.
- [8] mckinson, w. (2020). pandas: data manipulation and analysis.
- [9] Goodfellow, i. (2016). deep learning. mit press.
- [10] Hunter, j. d. (2007). matplotlib: a 2d graphics environment for python.
- [11] Waskin, m. (2016). seaborn: statistical data visualization with python.