Deep Learning :

Lab1:

Objective : The main purpose behind this lab is to get familiar with Pytorch library to do Classification and Regression tasks by establishing DNN/MLP architectures.

REALISE PAR:

DAMIATI KAOUTAR

Part 1 :

#import libraries

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

Charger les données

prices = pd.read_csv(r'C:\Users\damia\Downloads\nyse dataset\prices.csv')

prices_split_adjusted = pd.read_csv(r'C:\Users\damia\Downloads\nyse dataset\prices-split-adjusted.csv')

securities = pd.read_csv(r'C:\Users\damia\Downloads\nyse dataset\securities.csv')

fundamentals = pd.read_csv(r'C:\Users\damia\Downloads\nyse dataset\fundamentals.csv')

Afficher les premières lignes du fichier prices

print(prices.head())

Afficher les premières lignes du fichier prices_split_adjusted

print(prices_split_adjusted.head())

Afficher les premières lignes du fichier fundamentals

print(fundamentals.head())

Afficher les premières lignes du fichier securities

print(securities.head())

```
                  date symbol        open       close         low
high  \
0  2016-01-05 00:00:00   WLTW  123.430000  125.839996  122.309998
126.250000
1  2016-01-06 00:00:00   WLTW  125.239998  119.980003  119.940002
125.540001
2  2016-01-07 00:00:00   WLTW  116.379997  114.949997  114.930000
119.739998
3  2016-01-08 00:00:00   WLTW  115.480003  116.620003  113.500000
117.440002
4  2016-01-11 00:00:00   WLTW  117.010002  114.970001  114.089996
117.330002

      volume
0  2163600.0
1  2386400.0
2  2489500.0
3  2006300.0
4  1408600.0
```

Vérifier les valeurs manquantes pour les 4 fichiers

print("Missing values in Prices:")

print(prices.isnull().sum())

```
print("Missing values in Prices_split_adjusted:")
print(prices_split_adjusted.isnull().sum())


print("\nMissing values in Securities:")
print(securities.isnull().sum())


print("\nMissing values in Fundamentals:")
print(fundamentals.isnull().sum())
```

```
Missing values in Fundamentals:
Unnamed: 0                        0
Ticker Symbol                     0
Period Ending                     0
Accounts Payable                  0
Accounts Receivable               0
                               ...
Total Revenue                     0
Treasury Stock                    0
For Year                        173
Earnings Per Share              219
Estimated Shares Outstanding    219
Length: 79, dtype: int64
```

```
securities = securities.drop(columns=['Date first added'])
fundamentals = fundamentals.drop(columns=['For Year', 'Earnings Per Share', 'Estimated Shares Outstanding'])
# Afficher les statistiques descriptives
print("Statistics for Prices:")
print(prices.describe())


print("Statistics for Prices_split_adjusted:")
print(prices_split_adjusted.describe())


print("\nStatistics for Securities:")
print(securities.describe())


print("\nStatistics for Fundamentals:")
print(fundamentals.describe())
```

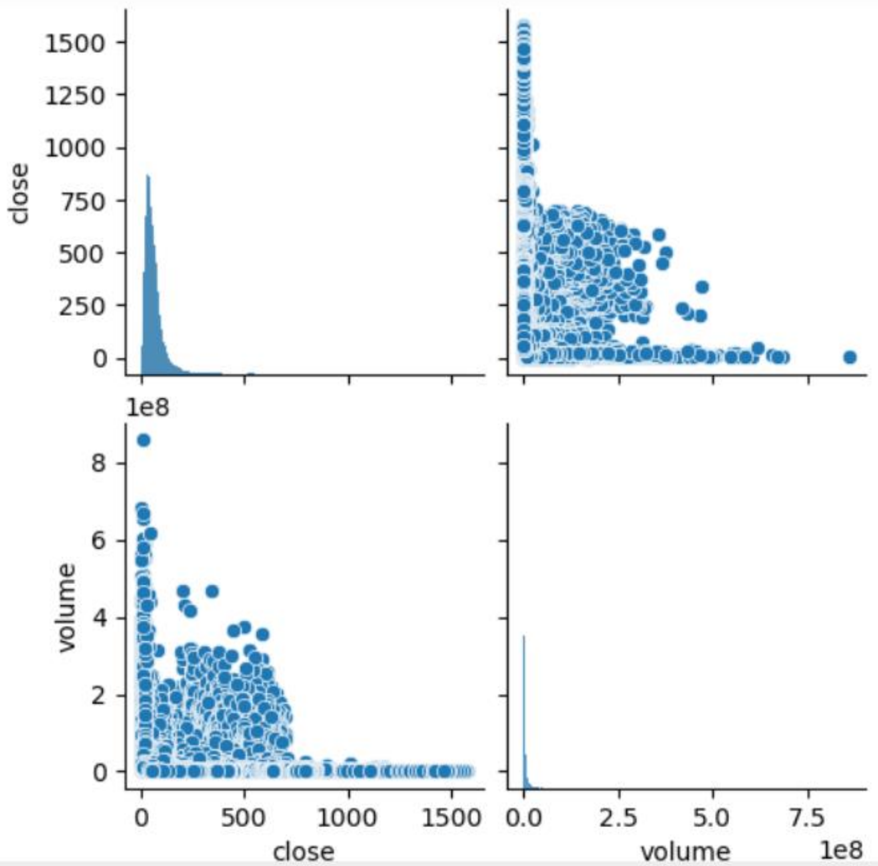
```
Statistics for Prices:
                open          close            low           high  \
count  851264.000000  851264.000000  851264.000000  851264.000000
mean       70.836986      70.857109      70.118414      71.543476
std        83.695876      83.689686      82.877294      84.465504
min         0.850000       0.860000       0.830000       0.880000
25%        33.840000      33.849998      33.480000      34.189999
50%        52.770000      52.799999      52.230000      53.310001
75%        79.879997      79.889999      79.110001      80.610001
max      1584.439941    1578.130005    1549.939941    1600.930054

             volume
count  8.512640e+05
mean   5.415113e+06
std    1.249468e+07
min    0.000000e+00
25%    1.221500e+06
50%    2.476250e+06
```

import matplotlib.pyplot as plt

import seaborn as sns

sns.pairplot(prices[['symbol', 'close', 'volume']])

plt.show()



```
from numpy import vstack, sqrt
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from torch.utils.data import Dataset, DataLoader, random_split
from torch import Tensor
from torch.nn import Linear, ReLU, Module
from torch.optim import SGD
from torch.nn import MSELoss
from torch.nn.init import xavier_uniform_
from tqdm import tqdm
import torch
import matplotlib.pyplot as plt

# Définir la classe PricesDataset pour votre ensemble de données NYSE
class PricesDataset(Dataset):
    def __init__(self, path):
        df = read_csv(path)
        # Sélectionnez les colonnes nécessaires pour votre modèle
        self.X = df[['open', 'low', 'high', 'volume']].values.astype('float32')
        self.y = df['close'].values.astype('float32')
        self.y = self.y.reshape((len(self.y), 1))

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

    def __getitem__(self, idx):
        return [self.X[idx], self.y[idx]]

    def get_splits(self, n_test=0.33):
        test_size = round(n_test * len(self.X))
        train_size = len(self.X) - test_size
        return random_split(self, [train_size, test_size])

# Définir la classe MLP pour votre ensemble de données NYSE avec les nouvelles caractéristiques
class NYSEMLP(Module):
    def __init__(self, n_inputs):
        super(NYSEMLP, self).__init__()
```

```
        self.hidden1 = Linear(n_inputs, 10)
        xavier_uniform_(self.hidden1.weight)
        self.act1 = ReLU()
        self.hidden2 = Linear(10, 8)
        xavier_uniform_(self.hidden2.weight)
        self.act2 = ReLU()
        self.hidden3 = Linear(8, 1)
        xavier_uniform_(self.hidden3.weight)

    def forward(self, X):
        X = self.hidden1(X)
        X = self.act1(X)
        X = self.hidden2(X)
        X = self.act2(X)
        X = self.hidden3(X)
        return X

# Préparer les données
def prepare_data(path):
    dataset = PricesDataset(path)
    train, test = dataset.get_splits()
    train_dl = DataLoader(train, batch_size=32, shuffle=True)
    test_dl = DataLoader(test, batch_size=1024, shuffle=False)
    return train_dl, test_dl

# Entraîner le modèle
def train_model(train_dl, model):
    size = len(train_dl.dataset)
    criterion = MSELoss()
    optimizer = SGD(model.parameters(), lr=0.00001, momentum=0.9)  # Réduction du taux d'apprentissage

    train_losses = []
    test_losses = []

    for epoch in tqdm(range(10), desc='Training Epochs'):
        print(f"Epoch {epoch + 1}\n-------------------------------")
        for batch, (inputs, targets) in enumerate(train_dl):
```

```python
            optimizer.zero_grad()
            yhat = model(inputs)

            # Vérifiez si les prédictions contiennent des valeurs nan ou inf
            if torch.isnan(yhat).any() or torch.isinf(yhat).any():
                continue

            loss = criterion(yhat, targets)

            # Calculer les gradients
            loss.backward()

            # Clippez l'ensemble des gradients pour éviter les explosions de gradient
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

            optimizer.step()

            loss, current = loss.item(), batch * len(inputs)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

        # Ajouter la perte d'entraînement à la liste
        train_losses.append(loss)

        # Évaluer le modèle sur l'ensemble de test
        test_loss = evaluate_model(test_dl, model)
        # Ajouter la perte de test à la liste
        test_losses.append(test_loss)

    return train_losses, test_losses

# Évaluer le modèle
def evaluate_model(test_dl, model):
    predictions, actuals = list(), list()
    for i, (inputs, targets) in enumerate(test_dl):
        yhat = model(inputs).detach().numpy()
        actual = targets.numpy().reshape((len(targets), 1))
        predictions.append(yhat)
```

```
        actuals.append(actual)

    predictions, actuals = vstack(predictions), vstack(actuals)
    mse = mean_squared_error(actuals, predictions)
    return mse

# Faire une prédiction pour une nouvelle ligne de données
def predict(row, model):
    row = Tensor([row])
    yhat = model(row).detach().numpy()
    return yhat

# Préparer les données
train_dl, test_dl = prepare_data(r'C:\Users\DELL\Desktop\S3MasterMBD\Deep Learning\archive\prices.csv')
print(len(train_dl.dataset), len(test_dl.dataset))

# Définir le modèle
model = NYSEMLP(4)  # Le nombre d'entrées dépend du nombre de caractéristiques sélectionnées

# Entraîner le modèle
train_losses, test_losses = train_model(train_dl, model)

# Évaluer le modèle
mse = evaluate_model(test_dl, model)
print('MSE: %.3f, RMSE: %.3f' % (mse, sqrt(mse)))

# Faire une prédiction pour une nouvelle ligne de données
new_data_row = [123.45, 120.00, 125.50, 2500000]  # Ajoutez les valeurs appropriées pour vos caractéristiques
prediction = predict(new_data_row, model)
print('Predicted: %.3f' % prediction)

# Visualiser les graphiques
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.legend()
plt.show()
```

```
Training Epochs:   0%|
| 0/10 [00:00<?, ?it/s]

Epoch 1
-------------------------------
loss: 7909778391040.000000  [    0/570347]
loss: 9588809138176.000000  [   32/570347]
loss: 957055565824.000000  [   64/570347]
loss: 85836766904320.000000  [   96/570347]
loss: 9561666748416.000000  [  128/570347]
loss: 5891390701568.000000  [  160/570347]
loss: 122830259748864.000000  [  192/570347]
loss: 3874570633216.000000  [  224/570347]
loss: 5230973419520.000000  [  256/570347]
loss: 5610124869632.000000  [  288/570347]
loss: 23039431409664.000000  [  320/570347]
loss: 2522352451584.000000  [  352/570347]
loss: 6450069372928.000000  [  384/570347]
loss: 4718532755456.000000  [  416/570347]
```

```
from sklearn.model_selection import GridSearchCV

parameters = {'lr': [0.001, 0.01, 0.1],
       'optimizer': ['adam', 'sgd'],
       'epochs': [10, 20, 30]}

model = PricesDataset(input_size, hidden_size, output_size)

optimizer = optim.SGD(model.parameters(), lr=0.01)

grid_search = GridSearchCV(estimator=model, param_grid=parameters, cv=3)

def train_model(model, train_loader, criterion, optimizer, epochs):
  train_losses = []

  for epoch in tqdm(range(epochs), desc='Training Epochs'):
    model.train()
    running_loss = 0.0

    for batch_idx, (inputs, targets) in enumerate(train_loader):
      optimizer.zero_grad()
      outputs = model(inputs)
      loss = criterion(outputs, targets)
      loss.backward()
      optimizer.step()

      running_loss += loss.item()

    # Calculate and store the average training loss for the epoch
    average_loss = running_loss / len(train_loader)
```

```python
        train_losses.append(average_loss)
        print(f"Epoch {epoch + 1}/{epochs}, Loss: {average_loss:.6f}")
    return train_losses
# Entraîner le modèle avec les meilleurs paramètres
best_params = grid_search.best_params_
model = PricesDataset(input_size, hidden_size, output_size)
optimizer = optim.SGD(model.parameters(), lr=best_params['lr'])
criterion = nn.MSELoss()
train_losses = train_model(model, train_loader, criterion, optimizer, best_params['epochs'])
# Visualiser les graphiques
plt.plot(train_losses, label='Train Loss')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss Over Epochs')
plt.show()
# Visualiser les graphiques
plt.plot(train_losses, label='Train Loss')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss Over Epochs')
plt.show()
from torch.nn import Dropout, BatchNorm1d

# Modifier le modèle pour inclure des techniques de régularisation
class RegularizedModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, dropout_rate=0.5):
        super(RegularizedModel, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(p=dropout_rate)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
```

```python
        x = self.dropout(x)
        x = self.fc2(x)
        return x
# Entraîner et évaluer le modèle avec régularisation
regularized_model = RegularizedModel(input_size, hidden_size, output_size)

# Utilisez le même code d'entraînement et d'évaluation que précédemment
optimizer = optim.SGD(regularized_model.parameters(), lr=best_params['lr'])
train_losses_reg = train_model(regularized_model, train_loader, criterion, optimizer, best_params['epochs'])

# Visualisez les graphiques pour le modèle régularisé
plt.plot(train_losses_reg, label='Train Loss (Regularized)')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss Over Epochs (Regularized)')
plt.show()
```

Part 2:

```python
#import libraries
from numpy import vstack, argmax
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
from torch import Tensor
from torch.utils.data import Dataset, DataLoader, random_split
from torch.nn import Linear, ReLU, Softmax, Module, Dropout
from torch.optim import Adam
from torch.nn import CrossEntropyLoss
from torch.nn.init import kaiming_uniform_, xavier_uniform_
from tqdm import tqdm
import numpy as np
import torch
import plotly.graph_objects as go
class CSVDataset(Dataset):
    def __init__(self, path):
        df = read_csv(path, header=0)
```

```
        # Exclude non-numeric columns and the 'UID' column
        non_numeric_columns = ['Product ID', 'Type', 'Target', 'Failure Type']
        df_numeric = df.drop(columns=non_numeric_columns)
        self.X = df_numeric.values.astype('float32')

        # Encode non-numeric categorical columns
        label_encoders = {}
        for column in non_numeric_columns:
            label_encoders[column] = LabelEncoder()
            df[column] = label_encoders[column].fit_transform(df[column])

        # Assign the encoded 'Target' column to y
        self.y = df['Target'].values

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

    def __getitem__(self, idx):
        return [self.X[idx], self.y[idx]]

    def get_splits(self, n_test=0.33):
        test_size = round(n_test * len(self.X))
        train_size = len(self.X) - test_size
        return random_split(self, [train_size, test_size])
#model definition
class MLP(Module):
    def __init__(self, n_inputs):
        super(MLP, self).__init__()
        self.hidden1 = Linear(n_inputs, 10)
        kaiming_uniform_(self.hidden1.weight, nonlinearity='relu')
        self.act1 = ReLU()
        self.dropout1 = Dropout(0.2)
        self.hidden2 = Linear(10, 8)
        kaiming_uniform_(self.hidden2.weight, nonlinearity='relu')
        self.act2 = ReLU()
        self.hidden3 = Linear(8, 3)
```

```
        xavier_uniform_(self.hidden3.weight)
        self.act3 = Softmax(dim=1)

    def forward(self, X):
        X = self.hidden1(X)
        X = self.act1(X)
        X = self.dropout1(X)
        X = self.hidden2(X)
        X = self.act2(X)
        X = self.hidden3(X)
        X = self.act3(X)
        return X
def prepare_data(path):
    dataset = CSVDataset(path)
    train, test = dataset.get_splits()
    train_dl = DataLoader(train, batch_size=1024, shuffle=True)
    test_dl = DataLoader(test, batch_size=1024, shuffle=False)
    return train_dl, test_dl
class EarlyStopping:
    def __init__(self, patience=7, verbose=False, delta=0, path='checkpoint.pt', trace_func=print):
        self.patience = patience
        self.verbose = verbose
        self.counter = 0
        self.best_score = None
        self.early_stop = False
        self.val_loss_min = np.Inf
        self.delta = delta
        self.path = path
        self.trace_func = trace_func

    def __call__(self, val_loss, model):
        score = -val_loss

        if self.best_score is None:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
        elif score < self.best_score + self.delta:
```

```
            self.counter += 1
            self.trace_func(f'EarlyStopping counter: {self.counter} out of {self.patience}')
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
            self.counter = 0

    def save_checkpoint(self, val_loss, model):
        if self.verbose:
            self.trace_func(f'Validation loss decreased ({self.val_loss_min:.6f} --> {val_loss:.6f}).  Saving model ...')
        torch.save(model.state_dict(), self.path)
        self.val_loss_min = val_loss

number_epochs = 500
learning_rate = 0.01
loss_per_epoch = []
loss_per_epoch_validation = []
class EarlyStopping:
    def __init__(self, patience=7, verbose=False, delta=0, path='checkpoint.pt', trace_func=print):
        self.patience = patience
        self.verbose = verbose
        self.counter = 0
        self.best_score = None
        self.early_stop = False
        self.val_loss_min = np.Inf
        self.delta = delta
        self.path = path
        self.trace_func = trace_func

    def __call__(self, val_loss, model):
        score = -val_loss

        if self.best_score is None:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
```

```
        elif score < self.best_score + self.delta:
            self.counter += 1
            self.trace_func(f'EarlyStopping counter: {self.counter} out of {self.patience}')
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
            self.counter = 0

    def save_checkpoint(self, val_loss, model):
        if self.verbose:
            self.trace_func(f'Validation loss decreased ({self.val_loss_min:.6f} --> {val_loss:.6f}).  Saving model ...')
        torch.save(model.state_dict(), self.path)
        self.val_loss_min = val_loss

number_epochs = 500
learning_rate = 0.01
loss_per_epoch = []
loss_per_epoch_validation = []
# evaluate the model
def evaluate_model(test_dl, model):
    predictions, actuals = list(), list()
    for i, (inputs, targets) in enumerate(test_dl):
        yhat = model(inputs)
        yhat = yhat.detach().numpy()
        actual = targets.numpy()
        yhat = argmax(yhat, axis=1)
        actual = actual.reshape((len(actual), 1))
        yhat = yhat.reshape((len(yhat), 1))
        predictions.append(yhat)
        actuals.append(actual)
    predictions, actuals = vstack(predictions), vstack(actuals)
    acc = accuracy_score(actuals, predictions)
    return acc

#make a class prediction for one row of data
```

def predict(row, model):

    row = Tensor([row])

    yhat = model(row)

    yhat = yhat.detach().numpy()

    return yhat

```
6700 3300
Training Epochs:   0%|                                              | 1/500 [00:00<01:26,  5.75it/s]
Epoch 1
-------------------------------
loss: 0.595945
Validation loss decreased (inf --> 0.584262).  Saving model ...
Epoch 2
-------------------------------
Training Epochs:   1%|▌                                             | 3/500 [00:00<00:59,  8.33it/s]
loss: 0.593538
Validation loss decreased (0.584262 --> 0.584262).  Saving model ...
Epoch 3
-------------------------------
loss: 0.588465
Validation loss decreased (0.584262 --> 0.584262).  Saving model ...
Epoch 4
```

```
Training Epochs:   1%|▌                                             | 4/500 [00:00<00:57,  8.61it/s]
loss: 0.587294
Validation loss decreased (0.584262 --> 0.584262).  Saving model ...
Epoch 5
-------------------------------
loss: 0.586377
Validation loss decreased (0.584262 --> 0.584262).  Saving model ...
Epoch 6
-------------------------------
loss: 0.587422
Training Epochs:   2%|█                                             | 8/500 [00:00<00:51,  9.63it/s]
Validation loss decreased (0.584262 --> 0.584262).  Saving model ...
Epoch 7
-------------------------------
loss: 0.586835
Validation loss decreased (0.584262 --> 0.584262).  Saving model ...
Epoch 8
```

Entrée [44]:
```
# evaluate the model
acc = evaluate_model(test_dl, model)
print('Accuracy: %.3f' % acc)
```

```
Accuracy: 0.969
```

#. loss function curve

import plotly.graph_objects as go

Assuming you have loss_per_epoch and loss_per_epoch_validation lists

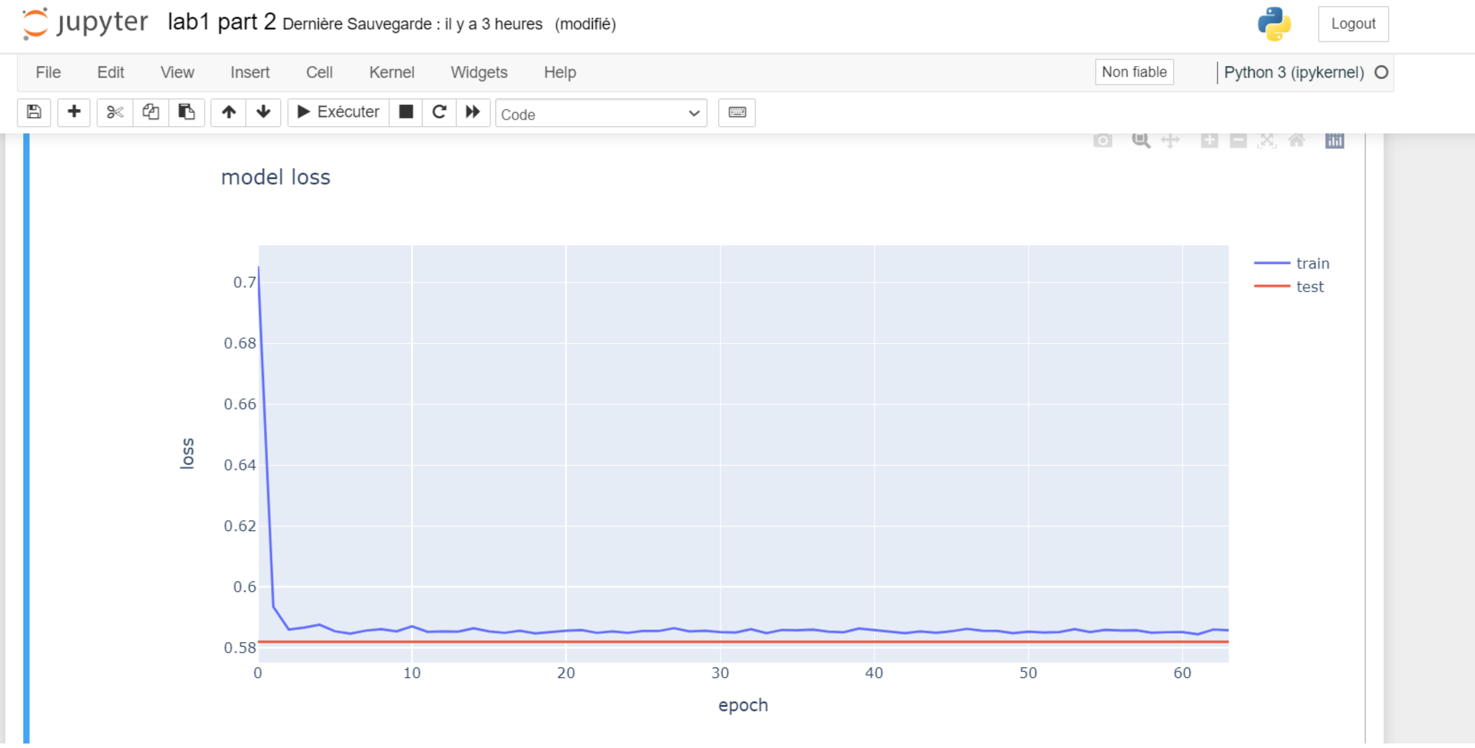
#loss_per_epoch = [0.5, 0.4, 0.3] # Replace with your actual list

#loss_per_epoch_validation = [0.6, 0.5, 0.4] # Replace with your actual list

Create the figure

fig = go.Figure()

```
# Add training loss trace
fig.add_trace(go.Scatter(x=list(range(len(loss_per_epoch))),
                         y=loss_per_epoch,
                         mode='lines',
                         name='train'))

# Add validation loss trace
fig.add_trace(go.Scatter(x=list(range(len(loss_per_epoch_validation))),
                         y=loss_per_epoch_validation,
                         mode='lines',
                         name='test'))

# Add labels and title
fig.update_layout(title='model loss',
                  xaxis=dict(title='epoch'),
                  yaxis=dict(title='loss'))

# Show the figure
fig.show()
#fig.write_image("LOSS_DNN_(30,20,10,1).svg")
```



Entrée [25]:
```
# make a single prediction
row = [51,298.9,309.1,2861,4.6,143]  # Replace with your actual input features
yhat = predict(row, model)
print('Predicted: %s (class=%d)' % (yhat, argmax(yhat)))
```

```
Predicted: [[1.000000e+00 5.161137e-08 3.050285e-10]] (class=0)
```

Entrée [ ]: