Homework 2

Name: Syeda Mishra Saiara
uID: u1457424

Assignment 1: Malware Detection using MLP and Drebin Dataset

The goal of this assignment is to train a multi layer perceptron model to detect android malware using the given Drebin dataset.

Firstly, the dataset was uploaded and split into 80-20 parts for training and test dataset.

```
# load dataset
#filepath = os.path.join('/data/', 'drebin_data.npz')
from google.colab import files
data = np.load('drebin_data.npz')
#data = np.load(filepath)
X, y = data['X'], data['y']
print(X.shape)
print(y.shape)
```

```
(3183, 1340)
(3183,)
```

```
# split into training and testing set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
[ ] print(f"X_train shape: {X_train.shape}")
    print(f"X_test shape: {X_test.shape}")
    print(f"y_train shape: {y_train.shape}")
    print(f"y_test shape: {y_test.shape}")
```

```
X_train shape: (2546, 1340)
X_test shape: (637, 1340)
y_train shape: (2546,)
y_test shape: (637,)
```

Next, an MLP model was implemented with input layers, hidden layers and output layers. First hidden layer takes the input size and maps it to 128 neurons. Then the second hidden layer reduces the size to 64 neurons and the third one to 32 neurons. There is an activation layer ReLU after each hidden layer to introduce non linearity.
The output layer first maps the 32 neurons to 7 classes. The softmax function activation layer ensures the output is a probability distribution over the 7 classes.

```python
class MLP(nn.Module):
    def __init__(self, input_size):
        super(MLP, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(input_size, 128),
            # define some middle layers
            nn.ReLU(),  # Activation function
            nn.Linear(128, 64),  # Hidden layer 1
            nn.ReLU(),  # Activation function
            nn.Linear(64, 32),  # Hidden layer 2
            nn.ReLU(),  # Activation function
            nn.Linear(32, 7),  # Output layer with 7 classes
            nn.Softmax(dim=1)  # Softmax activation for classification

        )

    def forward(self, x):
        return self.layers(x)
```

+ Code + Text

```python
# Data Preparation(may convert them into tensors)
X_train = torch.Tensor(X_train)  # Converting X_train to a PyTorch tensor
y_train = torch.Tensor(y_train).long().squeeze()  # Converting y_train to a PyTorch tensor, and squeezing to remove extra dimensions
X_test = torch.Tensor(X_test)  # Convert X_test to a PyTorch tensor
y_test = torch.Tensor(y_test).long().squeeze()  # Converting y_test to a PyTorch tensor and squeezing to remove extra dimensions

print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
torch.Size([2546, 1340])
torch.Size([2546])
torch.Size([637, 1340])
torch.Size([637])
```

The model is then trained in mini batches, having iterations for updating model weights to minimise loss. Then the performance is evaluated by precision, recall and F1 score.

```python
# Define your loss, optimizer, and other hyper-parameters
batch_size = 64
epochs = 20
learning_rate = 0.001

input_size = X.shape[1]
model = MLP(input_size)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

```python
# Training
for epoch in range(epochs):
    model.train()
    for i in range(0, X_train.shape[0], batch_size):
       # Get a batch of training data
        X_batch = X_train[i:i+batch_size]  # Getting inputs for this batch
        y_batch = y_train[i:i+batch_size]  # Getting corresponding labels for this batch

        # Convert to PyTorch tensors
        X_batch = torch.Tensor(X_batch)
        y_batch = torch.Tensor(y_batch).long()

        if len(y_batch.shape) > 1:  # If y_batch has more than 1 dimension
            y_batch = y_batch[:,0]  # Selecting the first element from the second dimension

         # Check if batch is smaller than batch_size
        if len(X_batch) < batch_size:
            continue # Skiping the batch if it's smaller than batch_size

        if X_batch.shape[0] != y_batch.shape[0]:
            # Skiping this iteration if there's a mismatch
            continue

        # Forward pass: compute predicted outputs by passing inputs to the model
        outputs = model(X_batch)
```

```python
    # Testing loss
    model.eval()
    with torch.no_grad():
        test_outputs = model(X_test)
        test_loss = criterion(test_outputs, y_test)

        predictions = torch.argmax(test_outputs, dim=1)
        accuracy = (predictions == y_test).float().mean()

    print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item()}, Test Loss: {test_loss.item()}, Test Acc: {accuracy}")
```

```
Epoch 1/20, Loss: 1.4445796012878418, Test Loss: 1.4221535921096802, Test Acc: 0.7598116397857666
Epoch 2/20, Loss: 1.2314707040786743, Test Loss: 1.2244514226913452, Test Acc: 0.9780219793319702
Epoch 3/20, Loss: 1.1837626695632935, Test Loss: 1.1803070306777954, Test Acc: 0.9890109896659851
Epoch 4/20, Loss: 1.1787784099578857, Test Loss: 1.175390362739563, Test Acc: 0.9921507239341736
Epoch 5/20, Loss: 1.1682156324386597, Test Loss: 1.174363374710083, Test Acc: 0.9937205910682678
Epoch 6/20, Loss: 1.1670653820037842, Test Loss: 1.170760154724121, Test Acc: 0.9984301328659058
Epoch 7/20, Loss: 1.1664751768112183, Test Loss: 1.1696022748947144, Test Acc: 0.9968602657318115
Epoch 8/20, Loss: 1.1657682657241821, Test Loss: 1.1694778203964233, Test Acc: 0.9968602657318115
Epoch 9/20, Loss: 1.1656917333602905, Test Loss: 1.1693884134292603, Test Acc: 0.9968602657318115
Epoch 10/20, Loss: 1.165639877319336, Test Loss: 1.1692655086517334, Test Acc: 0.9968602657318115
Epoch 11/20, Loss: 1.165602684020996, Test Loss: 1.1691796779632568, Test Acc: 0.9968602657318115
Epoch 12/20, Loss: 1.1655741930007935, Test Loss: 1.169114112854004, Test Acc: 0.9968602657318115
Epoch 13/20, Loss: 1.1655527353286743, Test Loss: 1.169061303138733, Test Acc: 0.9968602657318115
Epoch 14/20, Loss: 1.165535569190979, Test Loss: 1.169015884399414, Test Acc: 0.9968602657318115
Epoch 15/20, Loss: 1.1655209064483643, Test Loss: 1.168982744216919, Test Acc: 0.9968602657318115
Epoch 16/20, Loss: 1.1655092239379883, Test Loss: 1.1689536571502686, Test Acc: 0.9968602657318115
Epoch 17/20, Loss: 1.165499210357666, Test Loss: 1.168931007385254, Test Acc: 0.9968602657318115
Epoch 18/20, Loss: 1.1654900312423706, Test Loss: 1.1689162254333496, Test Acc: 0.9968602657318115
Epoch 19/20, Loss: 1.1654810905456543, Test Loss: 1.1689164638519287, Test Acc: 0.9968602657318115
Epoch 20/20, Loss: 1.1654691696166992, Test Loss: 1.1689621210098267, Test Acc: 0.9968602657318115
```

```python
# Calculate precision, recall, and F1 score for each class
precision = precision_score(y_test, predicted, average=None)
recall = recall_score(y_test, predicted, average=None)
f1 = f1_score(y_test, predicted, average=None)

# Create a 3x7 table (3 metrics for 7 classes)
metrics_table = np.vstack((precision, recall, f1))

# Display the metrics table
class_labels = np.arange(7)  # Assuming classes are labeled from 0 to 6
print("Metrics Table (Rows: Precision, Recall, F1 Score; Columns: Classes 0
print(metrics_table)

# Formatting and print as a DataFrame for better readability
import pandas as pd

metrics_df = pd.DataFrame(metrics_table, index=["Precision", "Recall", "F1 S
print(metrics_df)
```

```
Metrics Table (Rows: Precision, Recall, F1 Score; Columns: Classes 0 to 6):
[[0.99438202 1.         1.         0.98245614 1.         1.
  1.        ]
 [1.         0.98529412 1.         1.         1.         1.
  1.        ]
 [0.9971831  0.99259259 1.         0.99115044 1.         1.
  1.        ]]
                0         1    2         3    4    5    6
Precision  0.994382  1.000000  1.0  0.982456  1.0  1.0  1.0
Recall     1.000000  0.985294  1.0  1.000000  1.0  1.0  1.0
F1 Score   0.997183  0.992593  1.0  0.991150  1.0  1.0  1.0
```

Result explanations:

The model has a test loss of 1.17 and the test accuracy is 99% after the last epoch. And the precision, recall and F1 score of each 7 classes were also shown in a metrics table.

Assignment 2: Recognizing Functions in Binaries using RNN

The goal of the task is to detect function boundaries in sequences of bytes that are extracted from binary files using RNN.
Firstly, the dataset was loaded and preprocessed. First the sequences are converted to tensors, and then padded or truncated keeping the fixed length to 200.

```python
# load dataset
train_file = 'elf_x86_32_gcc_O1_train.pkl'
test_file = 'elf_x86_32_gcc_O1_test.pkl'

with open(train_file, 'rb') as f:
    x_train, y_train = pickle.load(f)

with open(test_file, 'rb') as f:
    x_test, y_test = pickle.load(f)
```

```python
# Convert the sequences to PyTorch tensors
x_train = [torch.tensor(seq) for seq in x_train]
x_test = [torch.tensor(seq) for seq in x_test]

# Set a fixed length for padding/truncating
fixed_length = 200

# Pad sequences to the fixed length
x_train_padded = pad_sequence(
    [seq[:fixed_length] for seq in x_train],  # Truncate if longer
    batch_first=True,
    padding_value=0  # Use 0 for padding
)

x_test_padded = pad_sequence(
    [seq[:fixed_length] for seq in x_test],
    batch_first=True,
    padding_value=0
)

y_train_padded = pad_sequence(
    [torch.tensor(seq[:fixed_length]) for seq in y_train],
    batch_first=True,
    padding_value=0
)

y_test_padded = pad_sequence(
    [torch.tensor(seq[:fixed_length]) for seq in y_test],
    batch_first=True,
    padding_value=0
)
```

```
# Checking the shapes of the prepared datasets
print(x_train_padded.shape)
print(y_train_padded.shape)
print(x_test_padded.shape)
print(y_test_padded.shape)
```

```
torch.Size([14006, 200])
torch.Size([14006, 200])
torch.Size([6003, 200])
torch.Size([6003, 200])
```

Then an RNN model was implemented using PyTorch. The architecture has an embedding layer, followed by an LSTM layer, a connected linear layer, and finally a Sigmoid activation function for binary classification.

```python
import torch
import torch.nn as nn
# Design you RNN model
class RNNModel(nn.Module):
    def __init__(self, seq_len, vocab_size, embed_dim, hidden_dim, num_layers, output_dim):
        super(RNNModel, self).__init__()
        super(RNNModel, self).__init__()
        # Embedding layer to learn a dense representation of the input bytes
        self.embedding = nn.Embedding(vocab_size, embed_dim)

        # LSTM layer
        self.lstm = nn.LSTM(embed_dim, hidden_dim, num_layers=num_layers, batch_first=True)
        # define some layers

         # Fully connected layer
        self.fc = nn.Linear(hidden_dim, output_dim)

        # Sigmoid activation for binary classification
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # Embedding the input sequence
        x = self.embedding(x)

        # Passing through the LSTM layer
        lstm_out, (hn, cn) = self.lstm(x)

        # Taking the output from the last time step (many-to-one)
        lstm_out_last = lstm_out[:, -1, :]  # shape: [batch_size, hidden_dim]

        # Passing through the fully connected layer
        fc_out = self.fc(lstm_out_last)
```

```python
[ ]         # Applying sigmoid activation
            out = self.sigmoid(fc_out)

            return out
        #def forward(self, x):
            # forward process

            #return x
```

```python
# Define your loss, optimizer, and other hyper-parameters
# Define loss function and optimizer
loss_fn = nn.BCELoss()  # Binary cross-entropy for binary classification
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)  # Adam optimizer

batch_size = 64
epochs = 10
learning_rate = 0.001

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
seq_len = 200
#model = RNNModel(seq_len).to(device)

# Define the RNN model
model = RNNModel(seq_len, vocab_size=256, embed_dim=128, hidden_dim=256, num_layers=2, output_dim=1).to(device)

# Define loss function (binary cross-entropy)
criterion = nn.BCELoss()

# Define optimizer (Adam optimizer)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

The model is then trained using binary cross entropy loss and optimized with the Adam optimizer.

```python
# Training
for epoch in range(epochs):
    model.train()  # Set the model to training mode
    total_loss = 0

    for i in range(0, x_train_padded.shape[0], batch_size):
        # Getting batch data
        batch_x = x_train_padded[i:i + batch_size].to(device)
        batch_y = y_train_padded[i:i + batch_size, -1].to(device).float()  # Ensuring labels a

        # Zero the gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = model(batch_x).squeeze()  # Squeezing for correct output shape for BCELoss

        # Compute the loss
        loss = criterion(outputs, batch_y)

        # Backpropagate the loss and update model's parameters
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    # Testing/Evaluation Phase
    model.eval()  # Set the model to evaluation mode
    test_loss = 0
    correct = 0
    total = 0
```

```python
    with torch.no_grad():
        for i in range(0, x_test_padded.shape[0], batch_size):
            batch_test_x = x_test_padded[i:i + batch_size].to(device)
            batch_test_y = y_test_padded[i:i + batch_size, -1].to(device).float()

            # Forward pass
            test_outputs = model(batch_test_x).squeeze()

            # Compute test loss
            loss_test = criterion(test_outputs, batch_test_y)
            test_loss += loss_test.item()

            # Calculate accuracy
            predictions = (test_outputs >= 0.5).float()  # Classifying based on threshold 0.5
            correct += (predictions == batch_test_y).sum().item()
            total += batch_test_y.size(0)
            accuracy = correct / total  # Calculate accuracy


    print(f"Epoch {epoch+1}/{epochs}, Loss: {total_loss / len(x_train_padded)}, Test Loss: {test_loss / le


! Also, don't forget to handle device assignments (to GPU or CPU) using .to(device) if you use GPU

! Save model
! torch.save(model.state_dict(), 'model_file.pth')
```

```
Epoch 1/10, Loss: 0.0002207953245181542, Test Loss: 0.0002381456573327, Test Acc: 0.9975012493753124
Epoch 2/10, Loss: 9.080215724067397e-05, Test Loss: 4.5113004164109494e-05, Test Acc: 0.9991670831251042
Epoch 3/10, Loss: 2.3109445364795758e-05, Test Loss: 3.7202013615927146e-05, Test Acc: 0.9995002498750625
Epoch 4/10, Loss: 1.600105870208751e-05, Test Loss: 3.768547315245874e-05, Test Acc: 0.9995002498750625
Epoch 5/10, Loss: 1.042347121166559e-05, Test Loss: 3.459051451716018e-05, Test Acc: 0.9998334166250208
Epoch 6/10, Loss: 1.2843809313828326e-05, Test Loss: 4.0087081728498824e-05, Test Acc: 0.9993336665000833
Epoch 7/10, Loss: 1.3152772601546962e-05, Test Loss: 3.9512627486435435e-05, Test Acc: 0.9991670831251042
Epoch 8/10, Loss: 9.097333365915057e-06, Test Loss: 3.632979087683282e-05, Test Acc: 0.9996668332500417
```

Lastly, the metrics were calculated for the model to evaluate its performance.

```python
# Convert predictions and labels to numpy arrays for metric calculations
all_preds = np.array(all_preds)
all_labels = np.array(all_labels)

# Calculate accuracy
accuracy = correct / total

# Calculate precision and recall
precision = precision_score(all_labels, all_preds)
recall = recall_score(all_labels, all_preds)

# Print results
print(f"Test Accuracy: {accuracy * 100:.2f}%")
print(f"Test Precision: {precision:.4f}")
print(f"Test Recall: {recall:.4f}")
```

```
Test Accuracy: 99.98%
Test Precision: 1.0000
Test Recall: 0.9333
```

Result explanation:

The test accuracy was 99% from the first epoch in this model. And the test loss was very significantly less in this case.
An observation from my side is that the model was too overfitted either due to the type of dataset or something wrong in preparing the model.

And the precision and recall were also close to 1.