CODE

```
import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader, Dataset

import numpy as np

from Bio import SeqIO

import os

import torch.nn.functional as F

from sklearn.metrics import classification_report, roc_auc_score, roc_curve,
confusion_matrix

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model_selection import train_test_split


# Set device

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


# File paths (Update these paths)

positive_file = "P.fasta"

negative_file = "NP.fasta"


# Maximum sequence length (512 in this case)

MAX_SEQ_LEN = 512


# Function to load sequences from fasta file, one-hot encode them, and pad to max length
```

```
def load_fasta_file(file):
    sequences = []
    for record in SeqIO.parse(file, "fasta"):
        seq = str(record.seq)
        one_hot_seq = aa_one_hot(seq)
        padded_seq = pad_sequence(one_hot_seq, MAX_SEQ_LEN)
        sequences.append(padded_seq)
    return sequences

# One-hot encoding function for amino acid sequences
def aa_one_hot(seq):
    amino_acids = 'ACDEFGHIKLMNPQRSTVWY'
    mapping = {aa: [1 if i == j else 0 for j in range(20)] for i, aa in enumerate(amino_acids)}
    return np.array([mapping.get(aa, [0]*20) for aa in seq])

# Function to pad the sequence to a fixed length (MAX_SEQ_LEN)
def pad_sequence(seq, max_len):
    pad_size = max_len - len(seq)
    if pad_size > 0:
        seq = np.vstack([seq, np.zeros((pad_size, seq.shape[1]))])
    elif pad_size < 0:
        seq = seq[:max_len]
    return seq

# Load data and labels
data = load_fasta_file(positive_file)
```

```
labels = [1] * len(data)

non_degrading_data = load_fasta_file(negative_file)
data.extend(non_degrading_data)
labels.extend([0] * len(non_degrading_data))

# Convert to torch tensors
data_np = torch.tensor(np.array(data), dtype=torch.float32)
labels_np = torch.tensor(np.array(labels), dtype=torch.long)

# Custom Dataset class for DNA sequences
class DNADataset(Dataset):
  def __init__(self, data, labels):
    self.data = data
    self.labels = labels

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

  def __getitem__(self, idx):
    return self.data[idx], self.labels[idx]

# Convert data to numpy arrays for easier manipulation
data_np = np.array(data)  # Your already loaded data
labels_np = np.array(labels)  # Labels as a numpy array
```

```
# Perform stratified split to ensure 20% of PET-degrading enzymes are in the test set
train_data, test_data, train_labels, test_labels = train_test_split(
    data_np, labels_np, test_size=0.2, stratify=labels_np, random_state=42
)

# Convert split data back to torch tensors
train_data = torch.tensor(train_data, dtype=torch.float32)
train_labels = torch.tensor(train_labels, dtype=torch.long)
test_data = torch.tensor(test_data, dtype=torch.float32)
test_labels = torch.tensor(test_labels, dtype=torch.long)

# Create Dataset objects
train_dataset = DNADataset(train_data, train_labels)
test_dataset = DNADataset(test_data, test_labels)

# DataLoaders
batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# Define a 1D CNN model
class CNN1D(nn.Module):
    def __init__(self):
        super(CNN1D, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=20, out_channels=64, kernel_size=7, stride=1,
padding=3)
```

```
        self.conv2 = nn.Conv1d(64, 128, kernel_size=7, stride=1, padding=3)
        self.pool = nn.MaxPool1d(2)
        self.fc1 = nn.Linear(128 * 128, 100)
        self.fc2 = nn.Linear(100, 50)
        self.fc3 = nn.Linear(50, 2)

    def forward(self, x):
        x = x.transpose(1, 2)
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Initialize the model, criterion, and optimizer
model = CNN1D().to(device)

# Use class weights to handle class imbalance
class_weights = torch.tensor([1.0, len(non_degrading_data) / len(data)],
dtype=torch.float32).to(device)
criterion = nn.CrossEntropyLoss(weight=class_weights)
optimizer = torch.optim.Adam(model.parameters(), lr=0.0005, weight_decay=1e-5)

# Training loop
```

```
num_epochs = 80
train_losses = []
for epoch in range(num_epochs):
    epoch_loss = 0.0
    model.train()
    for sequences, labels in train_loader:
        sequences, labels = sequences.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(sequences)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
    epoch_loss /= len(train_loader)
    train_losses.append(epoch_loss)
    print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {epoch_loss:.4f}")

# Plot training loss
plt.plot(range(1, num_epochs + 1), train_losses)
plt.title("Training Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()

# Evaluation metrics
model.eval()
```

```
all_labels = []
all_preds = []
all_probs = []
with torch.no_grad():
    for sequences, labels in test_loader:
        sequences, labels = sequences.to(device), labels.to(device)
        outputs = model(sequences)
        probs = F.softmax(outputs, dim=1)[:, 1].cpu().numpy()  # Class probabilities
        _, predicted = torch.max(outputs, 1)
        all_labels.extend(labels.cpu().numpy())
        all_preds.extend(predicted.cpu().numpy())
        all_probs.extend(probs)

# Classification report
print(classification_report(all_labels, all_preds))

# Calculate ROC AUC score
roc_auc = roc_auc_score(all_labels, all_probs)
print(f"ROC AUC: {roc_auc:.4f}")

# Confusion matrix
cm = confusion_matrix(all_labels, all_preds)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Non-degrading', 'Degrading'], yticklabels=['Non-degrading', 'Degrading'])
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
```

```
plt.ylabel('Actual')
plt.show()

# ROC Curve
fpr, tpr, thresholds = roc_curve(all_labels, all_probs)
plt.plot(fpr, tpr, label=f'ROC AUC = {roc_auc:.4f}')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()

# Save testing set sequences, predicted labels, and true labels
with open("test_results.txt", 'w') as f:
    for idx in range(len(test_data)):
        seq = test_data[idx][0].cpu().numpy()  # One-hot encoded sequence
        true_label = all_labels[idx]
        predicted_label = all_preds[idx]
        f.write(f"Sequence {idx} - True Label: {true_label}, Predicted Label: {predicted_label}\n")
```