Complex PyTorch for Music Genre Classification

In [61]:
```
# Complex pytorch
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from complexPyTorch.complexLayers import *
from complexPyTorch.complexFunctions import *
from torch_geometric.nn import GCNConv
from torch_geometric.data import Data

# Plot
import matplotlib.pyplot as plt
import seaborn as sns
import time

# Load Data
import numpy as np
import json
import os
import math
import librosa
import pathlib
from scipy.spatial.distance import cdist
from torch.utils.data import Dataset
from sklearn.model_selection import train_test_split
import random

# MFCCS
from scipy.io import wavfile
import scipy.fftpack as fft
from scipy.signal import get_window
```

Data Preparation

In [31]:
```
DATASET_PATH = "Data/binary_data/train"
SAMPLE_RATE = 22050
TRACK_DURATION = 30 # measured in seconds
SAMPLES_PER_TRACK = SAMPLE_RATE * TRACK_DURATION
BATCH_SIZE = 32
NUM_EPOCHS = 10
```

In [32]:
```
genre_list = os.listdir(DATASET_PATH)
if '.DS_Store' in genre_list: genre_list.remove('.DS_Store')
genre_mappings = dict(zip(genre_list, range(len(genre_list))))
print(genre_mappings)
```

```
{'classical': 0, 'rock': 1}
```

MFCCS

In [70]:
```
class MusicFeatureExtractorComplex2:
    def __init__(self, FFT_size=2048, HOP_SIZE=512, mel_filter_num=13, dc
```

```
        self.FFT_size = FFT_size
        self.HOP_SIZE = HOP_SIZE
        self.mel_filter_num = mel_filter_num
        self.dct_filter_num = dct_filter_num
        self.epsilon = 1e-10  # Added to log to avoid log10(0)

    def normalize_audio(self, audio):
        audio = audio / np.max(np.abs(audio))
        return audio

    def frame_audio(self, audio):
        frame_num = int((len(audio) - self.FFT_size) / self.HOP_SIZE) + 1
        frames = np.zeros((frame_num, self.FFT_size))
        for n in range(frame_num):
            frames[n] = audio[n * self.HOP_SIZE: n * self.HOP_SIZE + self
        return frames

    def freq_to_mel(self, freq):
        return 2595.0 * np.log10(1.0 + freq / 700.0)

    def met_to_freq(self, mels):
        return 700.0 * (10.0 ** (mels / 2595.0) - 1.0)

    def get_filter_points(self, fmin, fmax, sample_rate):
        fmin_mel = self.freq_to_mel(fmin)
        fmax_mel = self.freq_to_mel(fmax)
        mels = np.linspace(fmin_mel, fmax_mel, num=self.mel_filter_num +
        freqs = self.met_to_freq(mels)
        return np.floor((self.FFT_size + 1) / sample_rate * freqs).astype

    def get_filters(self, filter_points):
        filters = np.zeros((len(filter_points) - 2, int(self.FFT_size / 2
        for n in range(len(filter_points) - 2):
            filters[n, filter_points[n]: filter_points[n + 1]] = np.linsp
            filters[n, filter_points[n + 1]: filter_points[n + 2]] = np.l
        return filters

    def dct(self):
        basis = np.empty((self.dct_filter_num, self.mel_filter_num))
        basis[0, :] = 1.0 / np.sqrt(self.mel_filter_num)
        samples = np.arange(1, 2 * self.mel_filter_num, 2) * np.pi / (2.0
        for i in range(1, self.dct_filter_num):
            basis[i, :] = np.cos(i * samples) * np.sqrt(2.0 / self.mel_fi
        return basis

    def get_mfcc_features(self, audio, sample_rate):
        audio = self.normalize_audio(audio)
        audio_framed = self.frame_audio(audio)
        window = get_window("hann", self.FFT_size, fftbins=True)
        audio_win = audio_framed * window
        audio_winT = np.transpose(audio_win)
        audio_fft = np.empty((int(1 + self.FFT_size // 2), audio_winT.sha
        for n in range(audio_fft.shape[1]):
            audio_fft[:, n] = fft.fft(audio_winT[:, n], axis=0)[:audio_ff
        audio_fft = np.transpose(audio_fft)
        mag_fft = np.square(np.abs(audio_fft))
        phase_fft = np.angle(audio_fft)
        freq_min = 0
        freq_high = sample_rate / 2
        filter_points, mel_freqs = self.get_filter_points(freq_min, freq_
```

```
        filters = self.get_filters(filter_points)
        audio_filtered = np.dot(filters, np.transpose(mag_fft))
        phase_filtered = np.dot(filters, np.transpose(phase_fft))
        audio_filtered = np.maximum(audio_filtered, self.epsilon)  # Repl
        audio_log = 10.0 * np.log10(audio_filtered)
        dct_filters = self.dct()
        cepstral_coefficents = np.dot(dct_filters, audio_log)
        phase_coefficents = np.dot(dct_filters, phase_filtered)
        return np.array([cepstral_coefficents]), np.array([phase_coeffice
```

In [84]:

```
class GenreDatasetMFCC(Dataset):

    def __init__(self, train_path, n_fft=2048, hop_length=512, num_segmen
        cur_path = pathlib.Path(train_path)
        self.files = []
        for i in list(cur_path.rglob("*.wav")):
            for j in range(num_segments):
                self.files.append([j, i])
        self.samples_per_segment = int(SAMPLES_PER_TRACK / num_segments)
        self.n_fft = n_fft
        self.hop_length = hop_length
        self.num_segments = num_segments
        self.mfcc_extractor = MusicFeatureExtractor(
            FFT_size=n_fft, HOP_SIZE=hop_length, mel_filter_num = mel_fil
        self.dct_filter_num = dct_filter_num
        self.training = training

    def apply_augmentations(self, signal):
        # Apply augmentations to the audio signal
        if random.random() < 0.5:
            signal = librosa.effects.pitch_shift(signal, sr=SAMPLE_RATE,
        if random.random() < 0.5:
            signal = librosa.effects.time_stretch(signal, rate=random.uni
        return signal

    def adjust_shape(self, sequence, max_sequence_length = 126):
        current_length = sequence.shape[2]
        if current_length < max_sequence_length:
            padding = np.zeros((1, 13, max_sequence_length - current_leng
            padded_sequence = np.concatenate((sequence, padding), axis=2)
        else:
            padded_sequence = sequence[:, :, :max_sequence_length]
        return padded_sequence

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

    def __getitem__(self, idx):
        cur_file = self.files[idx]
        d = cur_file[0]
        file_path = cur_file[1]
        target = genre_mappings[str(file_path).split("/")[3]]
        signal, sample_rate = librosa.load(file_path, sr=SAMPLE_RATE)
        start = self.samples_per_segment * d
        finish = start + self.samples_per_segment
        cur_signal = signal[start:finish]
        if self.training: cur_signal = self.apply_augmentations(cur_signa
        cur_mfcc = self.mfcc_extractor.get_mfcc_features(cur_signal, samp
        cur_mfcc = self.adjust_shape(cur_mfcc)
        return torch.tensor(cur_mfcc, dtype=torch.float32), target
```

```
class GenreDatasetPhaseMFCC2(GenreDatasetMFCC):

    def __init__(self, train_path, n_fft=2048, hop_length=512, num_segmen
        super().__init__(train_path, n_fft, hop_length, num_segments, mel
        self.mfcc_extractor = MusicFeatureExtractorComplex2(
            FFT_size=n_fft, HOP_SIZE=hop_length, mel_filter_num = mel_fil

    def __getitem__(self, idx):
        cur_file = self.files[idx]
        d = cur_file[0]
        file_path = cur_file[1]
        target = genre_mappings[str(file_path).split("/")[3]]
        signal, sample_rate = librosa.load(file_path, sr=SAMPLE_RATE)
        start = self.samples_per_segment * d
        finish = start + self.samples_per_segment
        cur_signal = signal[start:finish]
        if self.training: cur_signal = self.apply_augmentations(cur_signa
        cur_mfcc, cur_phase = self.mfcc_extractor.get_mfcc_features(cur_s
        cur_mfcc, cur_phase = self.adjust_shape(cur_mfcc), self.adjust_sh
        return torch.tensor(cur_mfcc, dtype=torch.float32), torch.tensor(
```

In [99]:

```
def train(model, device, train_loader, test_loader, optimizer, epoch, met
    model.train()
    total_loss = 0
    correct = 0
    total_samples = len(train_loader.dataset)
    start_time = time.time()

    for batch_idx, (data, data2, target) in enumerate(train_loader):
        data, data2, target = data.to(device), data2.to(device), target.t
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()

        if batch_idx % 10 == 0:
            batch_accuracy = 100. * correct / ((batch_idx + 1) * len(data
            print('Train Epoch: {:3} [{:6}/{:6} ({:3.0f}%)]\tLoss: {:.6f}
                epoch,
                batch_idx * len(data),
                total_samples,
                100. * batch_idx / len(train_loader),
                loss.item(),
                batch_accuracy)
            )

    end_time = time.time()
    epoch_times = metrics_dict['epoch_times']
    epoch_times.append(end_time - start_time)
    epoch_loss = total_loss / len(train_loader)
    epoch_accuracy = 100. * correct / total_samples
    train_losses = metrics_dict['train_losses']
    train_accuracies = metrics_dict['train_accuracies']
    train_losses.append(epoch_loss)
    train_accuracies.append(epoch_accuracy)
```

```
    print('Epoch {} - Time: {:.2f}s - Train Loss: {:.6f} - Train Accuracy

    # Evaluate on test data
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, data2, target in test_loader:
            data, data2, target = data.to(device), data2.to(device), targ
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    test_accuracy = 100. * correct / len(test_loader.dataset)
    test_losses = metrics_dict['test_losses']
    test_accuracies = metrics_dict['test_accuracies']
    test_losses.append(test_loss)
    test_accuracies.append(test_accuracy)
    print('Test Loss: {:.6f} - Test Accuracy: {:.2f}%\n'.format(test_loss
```

In [100…

```
train_dataset = GenreDatasetPhaseMFCC2("Data/binary_data/train/", n_fft=2
test_dataset = GenreDatasetPhaseMFCC2("Data/binary_data/test/", n_fft=204
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, shuffle
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, shuffle=F
```

1. Simple Graph Net (Only magnitude)

In [101…

```
class ComplexGraphNet(nn.Module):
    def __init__(self):
        super(ComplexGraphNet, self).__init__()
        self.gnn_layer = GCNConv(in_channels=126, out_channels=126, node_
        self.conv1 = ComplexConv2d(1, 10, 2, 1)
        self.bn  = ComplexBatchNorm2d(10)
        self.conv2 = ComplexConv2d(10, 20, 2, 1)
        self.fc1 = ComplexLinear(30*2*20, 500)
        self.fc2 = ComplexLinear(500, 3)

    def forward(self, x):  # Pass edge_index for GNN
        batch_size, _, num_nodes, node_size = x.size()
        edge_index = torch.tensor([[i, j] for i in range(num_nodes) for j
        x = x.view(-1, num_nodes, node_size)  # Reshape for batch process
        x = self.gnn_layer(x, edge_index)
        x = x.unsqueeze(1)

        x = x.type(torch.complex64)
        x = self.conv1(x)
        x = complex_relu(x)
        x = complex_max_pool2d(x, 2, 2)
        x = self.bn(x)
        x = self.conv2(x)
        x = complex_relu(x)
        x = complex_max_pool2d(x, 2, 2)
        x = x.view(batch_size, -1)  # Reshape back to batched form
        x = self.fc1(x)
        x = complex_relu(x)
        x = self.fc2(x)
```

```
        x = x.abs()
        x = F.log_softmax(x, dim=1)
        return x

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = ComplexGraphNet().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

metrics_dict_e1 = {
    'epoch_times': [],
    'train_losses': [],
    'train_accuracies': [],
    'test_losses': [],
    'test_accuracies': []
}

for epoch in range(NUM_EPOCHS):
    train(model,
          device,
          train_loader,
          test_loader,
          optimizer,
          epoch,
          metrics_dict_e1)

print("-"*100)
print("-"*100)
print("FINAL RESULTS:")
print("-"*100)
for key, value in metrics_dict_e1.items():
    print(f'{key}: {value}')
```

```
Train Epoch:   0 [     0/  1680 (  0%)] Loss: 1.207374  Accuracy: 9.38%
Train Epoch:   0 [   320/  1680 ( 19%)] Loss: 3.140594  Accuracy: 47.73%
Train Epoch:   0 [   640/  1680 ( 38%)] Loss: 0.562662  Accuracy: 58.33%
Train Epoch:   0 [   960/  1680 ( 58%)] Loss: 0.407064  Accuracy: 63.00%
Train Epoch:   0 [  1280/  1680 ( 77%)] Loss: 0.379776  Accuracy: 65.24%
Train Epoch:   0 [  1600/  1680 ( 96%)] Loss: 0.912574  Accuracy: 66.24%
Epoch 0 - Time: 50.96s - Train Loss: 0.947542 - Train Accuracy: 65.71%
Test Loss: 0.673101 - Test Accuracy: 50.00%

Train Epoch:   1 [     0/  1680 (  0%)] Loss: 0.616933  Accuracy: 56.25%
Train Epoch:   1 [   320/  1680 ( 19%)] Loss: 0.426740  Accuracy: 69.60%
Train Epoch:   1 [   640/  1680 ( 38%)] Loss: 0.440670  Accuracy: 71.73%
Train Epoch:   1 [   960/  1680 ( 58%)] Loss: 0.720287  Accuracy: 72.78%
Train Epoch:   1 [  1280/  1680 ( 77%)] Loss: 0.502164  Accuracy: 73.55%
Train Epoch:   1 [  1600/  1680 ( 96%)] Loss: 0.467386  Accuracy: 73.84%
Epoch 1 - Time: 47.39s - Train Loss: 0.529978 - Train Accuracy: 73.21%
Test Loss: 0.515175 - Test Accuracy: 76.56%

Train Epoch:   2 [     0/  1680 (  0%)] Loss: 0.471520  Accuracy: 84.38%
Train Epoch:   2 [   320/  1680 ( 19%)] Loss: 0.565825  Accuracy: 78.12%
Train Epoch:   2 [   640/  1680 ( 38%)] Loss: 0.485707  Accuracy: 77.23%
Train Epoch:   2 [   960/  1680 ( 58%)] Loss: 0.507819  Accuracy: 76.92%
Train Epoch:   2 [  1280/  1680 ( 77%)] Loss: 0.634011  Accuracy: 76.52%
Train Epoch:   2 [  1600/  1680 ( 96%)] Loss: 0.454199  Accuracy: 76.41%
Epoch 2 - Time: 48.91s - Train Loss: 0.511152 - Train Accuracy: 75.36%
Test Loss: 0.506172 - Test Accuracy: 77.19%

Train Epoch:   3 [     0/  1680 (  0%)] Loss: 0.590825  Accuracy: 68.75%
Train Epoch:   3 [   320/  1680 ( 19%)] Loss: 0.539938  Accuracy: 74.15%
Train Epoch:   3 [   640/  1680 ( 38%)] Loss: 0.597324  Accuracy: 74.70%
Train Epoch:   3 [   960/  1680 ( 58%)] Loss: 0.542759  Accuracy: 74.09%
Train Epoch:   3 [  1280/  1680 ( 77%)] Loss: 0.617779  Accuracy: 74.47%
Train Epoch:   3 [  1600/  1680 ( 96%)] Loss: 0.510139  Accuracy: 75.12%
Epoch 3 - Time: 50.23s - Train Loss: 0.530607 - Train Accuracy: 74.52%
Test Loss: 0.514174 - Test Accuracy: 74.38%

Train Epoch:   4 [     0/  1680 (  0%)] Loss: 0.434674  Accuracy: 75.00%
Train Epoch:   4 [   320/  1680 ( 19%)] Loss: 0.662463  Accuracy: 73.86%
Train Epoch:   4 [   640/  1680 ( 38%)] Loss: 0.495664  Accuracy: 76.49%
Train Epoch:   4 [   960/  1680 ( 58%)] Loss: 0.752355  Accuracy: 76.41%
Train Epoch:   4 [  1280/  1680 ( 77%)] Loss: 0.603482  Accuracy: 74.85%
Train Epoch:   4 [  1600/  1680 ( 96%)] Loss: 0.866591  Accuracy: 75.18%
Epoch 4 - Time: 51.23s - Train Loss: 0.531305 - Train Accuracy: 74.46%
Test Loss: 0.528887 - Test Accuracy: 74.38%

Train Epoch:   5 [     0/  1680 (  0%)] Loss: 0.305450  Accuracy: 90.62%
Train Epoch:   5 [   320/  1680 ( 19%)] Loss: 0.523899  Accuracy: 76.99%
Train Epoch:   5 [   640/  1680 ( 38%)] Loss: 0.537668  Accuracy: 76.19%
Train Epoch:   5 [   960/  1680 ( 58%)] Loss: 0.520323  Accuracy: 76.81%
Train Epoch:   5 [  1280/  1680 ( 77%)] Loss: 0.402039  Accuracy: 76.68%
Train Epoch:   5 [  1600/  1680 ( 96%)] Loss: 0.375898  Accuracy: 76.90%
Epoch 5 - Time: 51.59s - Train Loss: 0.505864 - Train Accuracy: 76.13%
Test Loss: 0.521493 - Test Accuracy: 75.31%

Train Epoch:   6 [     0/  1680 (  0%)] Loss: 0.708187  Accuracy: 75.00%
Train Epoch:   6 [   320/  1680 ( 19%)] Loss: 0.409191  Accuracy: 75.28%
Train Epoch:   6 [   640/  1680 ( 38%)] Loss: 0.567735  Accuracy: 76.64%
Train Epoch:   6 [   960/  1680 ( 58%)] Loss: 0.488376  Accuracy: 75.60%
Train Epoch:   6 [  1280/  1680 ( 77%)] Loss: 0.426539  Accuracy: 76.98%
Train Epoch:   6 [  1600/  1680 ( 96%)] Loss: 0.571441  Accuracy: 76.96%
```

```
Epoch 6 - Time: 50.14s - Train Loss: 0.500705 - Train Accuracy: 76.25%
Test Loss: 0.490693 - Test Accuracy: 76.56%

Train Epoch:   7 [     0/  1680 (  0%)] Loss: 0.489590  Accuracy: 78.12%
Train Epoch:   7 [   320/  1680 ( 19%)] Loss: 0.502673  Accuracy: 76.14%
Train Epoch:   7 [   640/  1680 ( 38%)] Loss: 0.480265  Accuracy: 75.30%
Train Epoch:   7 [   960/  1680 ( 58%)] Loss: 0.424492  Accuracy: 76.21%
Train Epoch:   7 [  1280/  1680 ( 77%)] Loss: 0.432237  Accuracy: 77.21%
Train Epoch:   7 [  1600/  1680 ( 96%)] Loss: 0.412701  Accuracy: 76.96%
Epoch 7 - Time: 50.00s - Train Loss: 0.498504 - Train Accuracy: 76.25%
Test Loss: 0.505527 - Test Accuracy: 74.06%

Train Epoch:   8 [     0/  1680 (  0%)] Loss: 0.538931  Accuracy: 68.75%
Train Epoch:   8 [   320/  1680 ( 19%)] Loss: 0.684731  Accuracy: 75.28%
Train Epoch:   8 [   640/  1680 ( 38%)] Loss: 0.433309  Accuracy: 74.85%
Train Epoch:   8 [   960/  1680 ( 58%)] Loss: 0.415918  Accuracy: 76.31%
Train Epoch:   8 [  1280/  1680 ( 77%)] Loss: 0.322066  Accuracy: 77.67%
Train Epoch:   8 [  1600/  1680 ( 96%)] Loss: 0.564485  Accuracy: 77.57%
Epoch 8 - Time: 48.14s - Train Loss: 0.496900 - Train Accuracy: 76.85%
Test Loss: 0.526901 - Test Accuracy: 74.69%

Train Epoch:   9 [     0/  1680 (  0%)] Loss: 0.629545  Accuracy: 65.62%
Train Epoch:   9 [   320/  1680 ( 19%)] Loss: 0.429591  Accuracy: 75.28%
Train Epoch:   9 [   640/  1680 ( 38%)] Loss: 0.435359  Accuracy: 74.26%
Train Epoch:   9 [   960/  1680 ( 58%)] Loss: 0.415464  Accuracy: 74.50%
Train Epoch:   9 [  1280/  1680 ( 77%)] Loss: 0.587628  Accuracy: 75.15%
Train Epoch:   9 [  1600/  1680 ( 96%)] Loss: 0.411037  Accuracy: 74.88%
Epoch 9 - Time: 48.92s - Train Loss: 0.514325 - Train Accuracy: 74.17%
Test Loss: 0.513260 - Test Accuracy: 76.88%

------------------------------------------------------------------------
-------------------------
------------------------------------------------------------------------
-------------------------
FINAL RESULTS:
------------------------------------------------------------------------
-------------------------
epoch_times: [50.9602210521698, 47.38573408126831, 48.910207986831665, 50.
232210874557495, 51.23161029815674, 51.58642268180847, 50.13818717002869,
50.00428891181946, 48.13930010795593, 48.91971302032471]
train_losses: [0.9475423275278165, 0.5299778718214768, 0.5111522009739509,
0.5306065890651482, 0.5313048718067316, 0.5058639267316232, 0.500704648976
1426, 0.4985038024874834, 0.49689960651672804, 0.5143245343978589]
train_accuracies: [65.71428571428571, 73.21428571428571, 75.3571428571428
6, 74.52380952380952, 74.46428571428571, 76.13095238095238, 76.25, 76.25,
76.8452380952381, 74.16666666666667]
test_losses: [0.6731009095907211, 0.515174612402916, 0.5061720326542855,
0.5141735032200814, 0.5288872435688973, 0.5214934915304184, 0.490693284571
1708, 0.5055271267890931, 0.5269006744027138, 0.5132596462965011]
test_accuracies: [50.0, 76.5625, 77.1875, 74.375, 74.375, 75.3125, 76.562
5, 74.0625, 74.6875, 76.875]
```

2. Simple Graph Net (Magniude + phase wieghts)

In [104…

```python
def train(model, device, train_loader, test_loader, optimizer, epoch, met
    model.train()
    total_loss = 0
    correct = 0
```

```
    total_samples = len(train_loader.dataset)
    start_time = time.time()

    for batch_idx, (data, data2, target) in enumerate(train_loader):
        data, data2, target = data.to(device), data2.to(device), target.t
        optimizer.zero_grad()
        output = model([data, data2])
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()

        if batch_idx % 10 == 0:
            batch_accuracy = 100. * correct / ((batch_idx + 1) * len(data
            print('Train Epoch: {:3} [{:6}/{:6} ({:3.0f}%)]\tLoss: {:.6f}
                epoch,
                batch_idx * len(data),
                total_samples,
                100. * batch_idx / len(train_loader),
                loss.item(),
                batch_accuracy)
            )

    end_time = time.time()
    epoch_times = metrics_dict['epoch_times']
    epoch_times.append(end_time - start_time)
    epoch_loss = total_loss / len(train_loader)
    epoch_accuracy = 100. * correct / total_samples
    train_losses = metrics_dict['train_losses']
    train_accuracies = metrics_dict['train_accuracies']
    train_losses.append(epoch_loss)
    train_accuracies.append(epoch_accuracy)
    print('Epoch {} - Time: {:.2f}s - Train Loss: {:.6f} - Train Accuracy

    # Evaluate on test data
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, data2, target in test_loader:
            data, data2, target = data.to(device), data2.to(device), targ
            output = model([data, data2])
            test_loss += F.nll_loss(output, target, reduction='sum').item
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    test_accuracy = 100. * correct / len(test_loader.dataset)
    test_losses = metrics_dict['test_losses']
    test_accuracies = metrics_dict['test_accuracies']
    test_losses.append(test_loss)
    test_accuracies.append(test_accuracy)
    print('Test Loss: {:.6f} - Test Accuracy: {:.2f}%\n'.format(test_loss
```

In [126…

```
class ComplexGraphNet(nn.Module):
    def __init__(self):
        super(ComplexGraphNet, self).__init__()
        self.gnn_layer = GCNConv(in_channels=126, out_channels=126, node_
```

```
        self.conv1 = ComplexConv2d(1, 10, 2, 1)
        self.bn  = ComplexBatchNorm2d(10)
        self.conv2 = ComplexConv2d(10, 20, 2, 1)
        self.fc1 = ComplexLinear(30*2*20, 500)
        self.fc2 = ComplexLinear(500, 3)

    def forward(self, x):  # Pass edge_index for GNN
        x, phase_data = x[0], x[1]
        batch_size, _, num_nodes, node_size = x.size()
        edge_index = torch.tensor([[i, j] for i in range(num_nodes) for j
        phase_data = torch.mean(phase_data.view(-1, num_nodes, node_size)
        edge_weight = torch.tensor([torch.mean(np.abs(phase_data[edge_ind
                                                    phase_data[edge_ind
                                    for i in range(len(edge_index[0]))])
        x = x.view(-1, num_nodes, node_size)  # Reshape for batch process
        x = self.gnn_layer(x, edge_index, edge_weight)
        x = x.unsqueeze(1)

        x = x.type(torch.complex64)
        x = self.conv1(x)
        x = complex_relu(x)
        x = complex_max_pool2d(x, 2, 2)
        x = self.bn(x)
        x = self.conv2(x)
        x = complex_relu(x)
        x = complex_max_pool2d(x, 2, 2)
        x = x.view(batch_size, -1)  # Reshape back to batched form
        x = self.fc1(x)
        x = complex_relu(x)
        x = self.fc2(x)
        x = x.abs()
        x = F.log_softmax(x, dim=1)
        return x

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = ComplexGraphNet().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

metrics_dict_e2 = {
    'epoch_times': [],
    'train_losses': [],
    'train_accuracies': [],
    'test_losses': [],
    'test_accuracies': []
}

for epoch in range(NUM_EPOCHS):
    train(model,
          device,
          train_loader,
          test_loader,
          optimizer,
          epoch,
          metrics_dict_e2)

print("-"*100)
print("-"*100)
print("FINAL RESULTS:")
print("-"*100)
```

```
for key, value in metrics_dict_e2.items():
    print(f'{key}: {value}')
```

```
Train Epoch:   0 [     0/  1680 (  0%)] Loss: 1.267587  Accuracy: 9.38%
Train Epoch:   0 [   320/  1680 ( 19%)] Loss: 1.406211  Accuracy: 39.49%
Train Epoch:   0 [   640/  1680 ( 38%)] Loss: 0.609435  Accuracy: 49.55%
Train Epoch:   0 [   960/  1680 ( 58%)] Loss: 0.501241  Accuracy: 57.26%
Train Epoch:   0 [  1280/  1680 ( 77%)] Loss: 0.418911  Accuracy: 62.58%
Train Epoch:   0 [  1600/  1680 ( 96%)] Loss: 0.817381  Accuracy: 61.34%
Epoch 0 - Time: 50.67s - Train Loss: 0.951244 - Train Accuracy: 61.01%
Test Loss: 0.605936 - Test Accuracy: 71.56%

Train Epoch:   1 [     0/  1680 (  0%)] Loss: 0.606225  Accuracy: 68.75%
Train Epoch:   1 [   320/  1680 ( 19%)] Loss: 0.458616  Accuracy: 81.25%
Train Epoch:   1 [   640/  1680 ( 38%)] Loss: 0.461616  Accuracy: 81.40%
Train Epoch:   1 [   960/  1680 ( 58%)] Loss: 0.379228  Accuracy: 81.15%
Train Epoch:   1 [  1280/  1680 ( 77%)] Loss: 0.691223  Accuracy: 81.17%
Train Epoch:   1 [  1600/  1680 ( 96%)] Loss: 0.471031  Accuracy: 81.56%
Epoch 1 - Time: 48.56s - Train Loss: 0.433627 - Train Accuracy: 80.71%
Test Loss: 0.480244 - Test Accuracy: 76.88%

Train Epoch:   2 [     0/  1680 (  0%)] Loss: 0.279182  Accuracy: 84.38%
Train Epoch:   2 [   320/  1680 ( 19%)] Loss: 0.270821  Accuracy: 87.22%
Train Epoch:   2 [   640/  1680 ( 38%)] Loss: 0.503920  Accuracy: 83.93%
Train Epoch:   2 [   960/  1680 ( 58%)] Loss: 0.282835  Accuracy: 83.06%
Train Epoch:   2 [  1280/  1680 ( 77%)] Loss: 0.327560  Accuracy: 83.38%
Train Epoch:   2 [  1600/  1680 ( 96%)] Loss: 0.212326  Accuracy: 82.90%
Epoch 2 - Time: 54.62s - Train Loss: 0.384960 - Train Accuracy: 82.32%
Test Loss: 0.415069 - Test Accuracy: 81.88%

Train Epoch:   3 [     0/  1680 (  0%)] Loss: 0.328330  Accuracy: 90.62%
Train Epoch:   3 [   320/  1680 ( 19%)] Loss: 0.374006  Accuracy: 84.09%
Train Epoch:   3 [   640/  1680 ( 38%)] Loss: 0.317814  Accuracy: 84.23%
Train Epoch:   3 [   960/  1680 ( 58%)] Loss: 0.334968  Accuracy: 84.78%
Train Epoch:   3 [  1280/  1680 ( 77%)] Loss: 0.334604  Accuracy: 84.83%
Train Epoch:   3 [  1600/  1680 ( 96%)] Loss: 0.398636  Accuracy: 84.74%
Epoch 3 - Time: 57.57s - Train Loss: 0.367431 - Train Accuracy: 83.99%
Test Loss: 0.582392 - Test Accuracy: 72.50%

Train Epoch:   4 [     0/  1680 (  0%)] Loss: 0.329880  Accuracy: 87.50%
Train Epoch:   4 [   320/  1680 ( 19%)] Loss: 0.262071  Accuracy: 83.24%
Train Epoch:   4 [   640/  1680 ( 38%)] Loss: 0.262595  Accuracy: 83.04%
Train Epoch:   4 [   960/  1680 ( 58%)] Loss: 0.320889  Accuracy: 84.48%
Train Epoch:   4 [  1280/  1680 ( 77%)] Loss: 0.424997  Accuracy: 85.21%
Train Epoch:   4 [  1600/  1680 ( 96%)] Loss: 0.368613  Accuracy: 85.72%
Epoch 4 - Time: 51.98s - Train Loss: 0.331098 - Train Accuracy: 85.00%
Test Loss: 0.331612 - Test Accuracy: 87.81%

Train Epoch:   5 [     0/  1680 (  0%)] Loss: 0.410566  Accuracy: 84.38%
Train Epoch:   5 [   320/  1680 ( 19%)] Loss: 0.284102  Accuracy: 85.51%
Train Epoch:   5 [   640/  1680 ( 38%)] Loss: 0.372883  Accuracy: 84.82%
Train Epoch:   5 [   960/  1680 ( 58%)] Loss: 0.424838  Accuracy: 85.28%
Train Epoch:   5 [  1280/  1680 ( 77%)] Loss: 0.334458  Accuracy: 85.21%
Train Epoch:   5 [  1600/  1680 ( 96%)] Loss: 0.170076  Accuracy: 86.03%
Epoch 5 - Time: 45.13s - Train Loss: 0.335689 - Train Accuracy: 85.18%
Test Loss: 0.337801 - Test Accuracy: 86.56%

Train Epoch:   6 [     0/  1680 (  0%)] Loss: 0.183198  Accuracy: 93.75%
Train Epoch:   6 [   320/  1680 ( 19%)] Loss: 0.256637  Accuracy: 82.95%
Train Epoch:   6 [   640/  1680 ( 38%)] Loss: 0.657049  Accuracy: 83.04%
Train Epoch:   6 [   960/  1680 ( 58%)] Loss: 0.256629  Accuracy: 84.58%
Train Epoch:   6 [  1280/  1680 ( 77%)] Loss: 0.514066  Accuracy: 84.07%
Train Epoch:   6 [  1600/  1680 ( 96%)] Loss: 0.312152  Accuracy: 84.56%
```

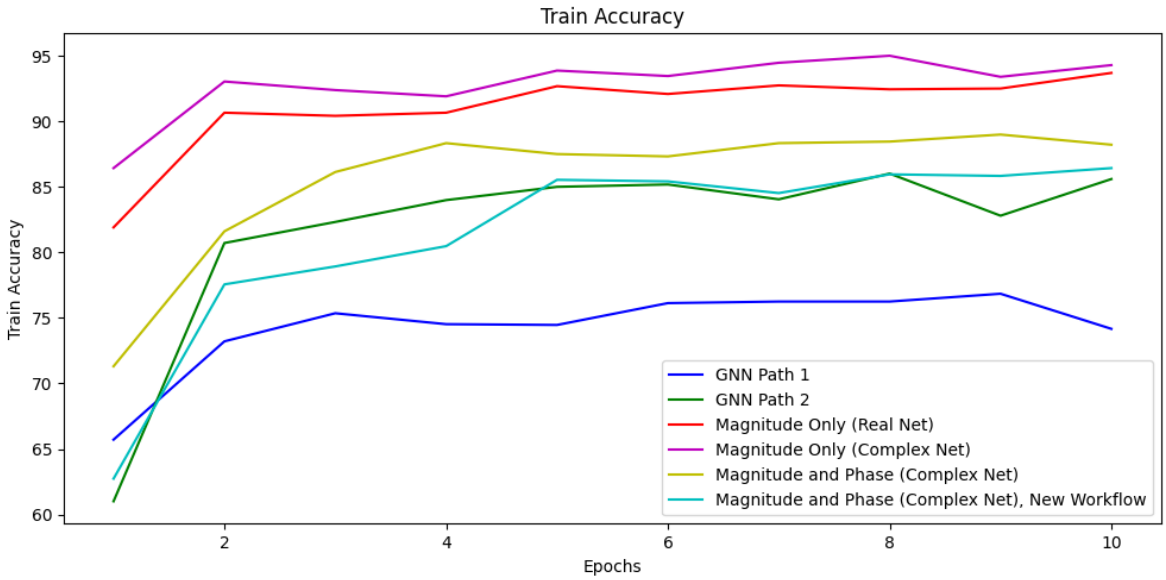
```
Epoch 6 - Time: 45.78s - Train Loss: 0.343581 - Train Accuracy: 84.05%
Test Loss: 0.235073 - Test Accuracy: 90.94%

Train Epoch:   7 [     0/  1680 (  0%)] Loss: 0.283756  Accuracy: 81.25%
Train Epoch:   7 [   320/  1680 ( 19%)] Loss: 0.266070  Accuracy: 82.67%
Train Epoch:   7 [   640/  1680 ( 38%)] Loss: 0.418715  Accuracy: 84.08%
Train Epoch:   7 [   960/  1680 ( 58%)] Loss: 0.618876  Accuracy: 84.98%
Train Epoch:   7 [  1280/  1680 ( 77%)] Loss: 0.227366  Accuracy: 86.05%
Train Epoch:   7 [  1600/  1680 ( 96%)] Loss: 0.172797  Accuracy: 87.13%
Epoch 7 - Time: 44.61s - Train Loss: 0.311824 - Train Accuracy: 86.01%
Test Loss: 0.306361 - Test Accuracy: 88.75%

Train Epoch:   8 [     0/  1680 (  0%)] Loss: 0.177947  Accuracy: 93.75%
Train Epoch:   8 [   320/  1680 ( 19%)] Loss: 0.240810  Accuracy: 82.95%
Train Epoch:   8 [   640/  1680 ( 38%)] Loss: 0.418358  Accuracy: 81.99%
Train Epoch:   8 [   960/  1680 ( 58%)] Loss: 0.301184  Accuracy: 83.27%
Train Epoch:   8 [  1280/  1680 ( 77%)] Loss: 0.730459  Accuracy: 82.77%
Train Epoch:   8 [  1600/  1680 ( 96%)] Loss: 0.496752  Accuracy: 83.52%
Epoch 8 - Time: 46.03s - Train Loss: 0.374520 - Train Accuracy: 82.80%
Test Loss: 0.279294 - Test Accuracy: 89.69%

Train Epoch:   9 [     0/  1680 (  0%)] Loss: 0.542186  Accuracy: 84.38%
Train Epoch:   9 [   320/  1680 ( 19%)] Loss: 0.206770  Accuracy: 84.66%
Train Epoch:   9 [   640/  1680 ( 38%)] Loss: 0.311039  Accuracy: 86.46%
Train Epoch:   9 [   960/  1680 ( 58%)] Loss: 0.203693  Accuracy: 85.69%
Train Epoch:   9 [  1280/  1680 ( 77%)] Loss: 0.603102  Accuracy: 85.90%
Train Epoch:   9 [  1600/  1680 ( 96%)] Loss: 0.261831  Accuracy: 86.34%
Epoch 9 - Time: 47.31s - Train Loss: 0.336600 - Train Accuracy: 85.60%
Test Loss: 0.420695 - Test Accuracy: 80.31%

-------------------------------------------------------------------------
-------------------------
-------------------------------------------------------------------------
-------------------------
FINAL RESULTS:
-------------------------------------------------------------------------
-------------------------
epoch_times: [50.66547727584839, 48.55875086784363, 54.617011308670044, 5
7.566506147384644, 51.984057903289795, 45.127379179000854, 45.784584045410
156, 44.609222173690796, 46.03325295448303, 47.30795168876648]
train_losses: [0.9512442281612983, 0.4336274601519108, 0.384960412979126,
0.36743127525999, 0.33109788539317936, 0.3356893973854872, 0.3435813418517
8923, 0.31182445069918263, 0.3745197103573726, 0.336599862203002]
train_accuracies: [61.01190476190476, 80.71428571428571, 82.3214285714285
7, 83.98809523809524, 85.0, 85.17857142857143, 84.04761904761905, 86.01190
476190476, 82.79761904761905, 85.5952380952381]
test_losses: [0.6059358954429627, 0.48024370297789576, 0.4150694083422422
6, 0.5823921039700508, 0.33161151092499497, 0.3378014832735062, 0.23507254
477590322, 0.3063611211255193, 0.27929382198490205, 0.42069460917264223]
test_accuracies: [71.5625, 76.875, 81.875, 72.5, 87.8125, 86.5625, 90.937
5, 88.75, 89.6875, 80.3125]
```

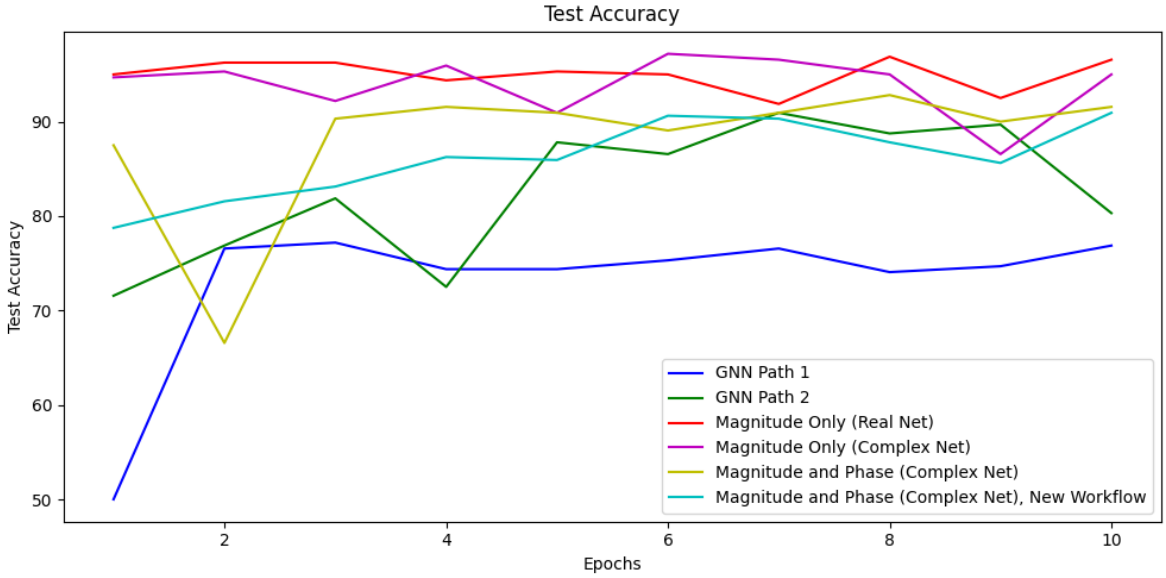
Plots

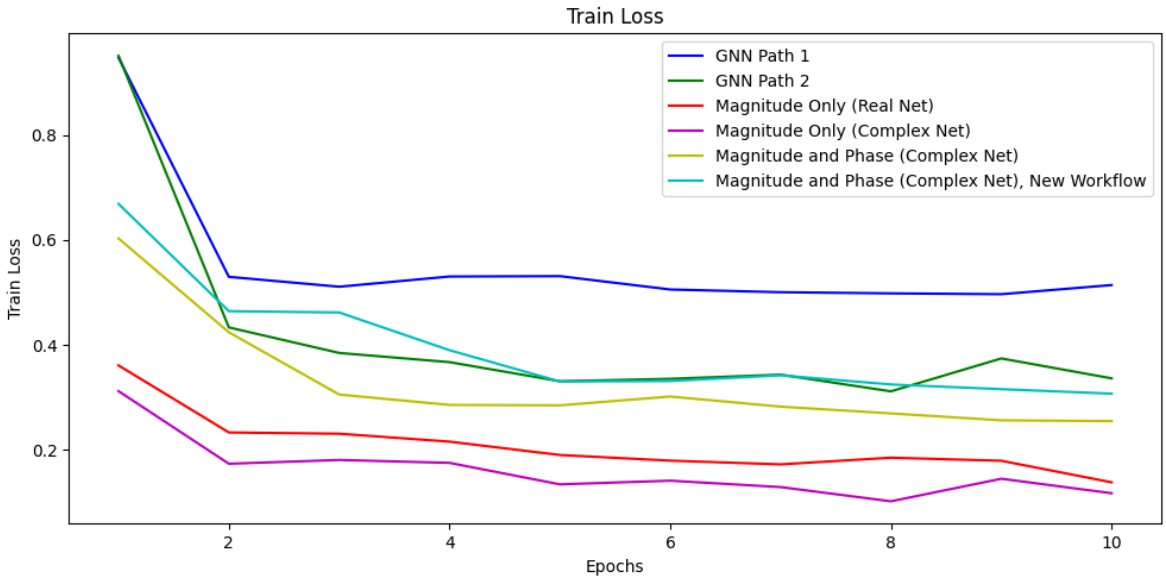
In [127…
```
data_old = {'Magnitude Only (Real Net)': {'epoch_times': [37.517344951629
```

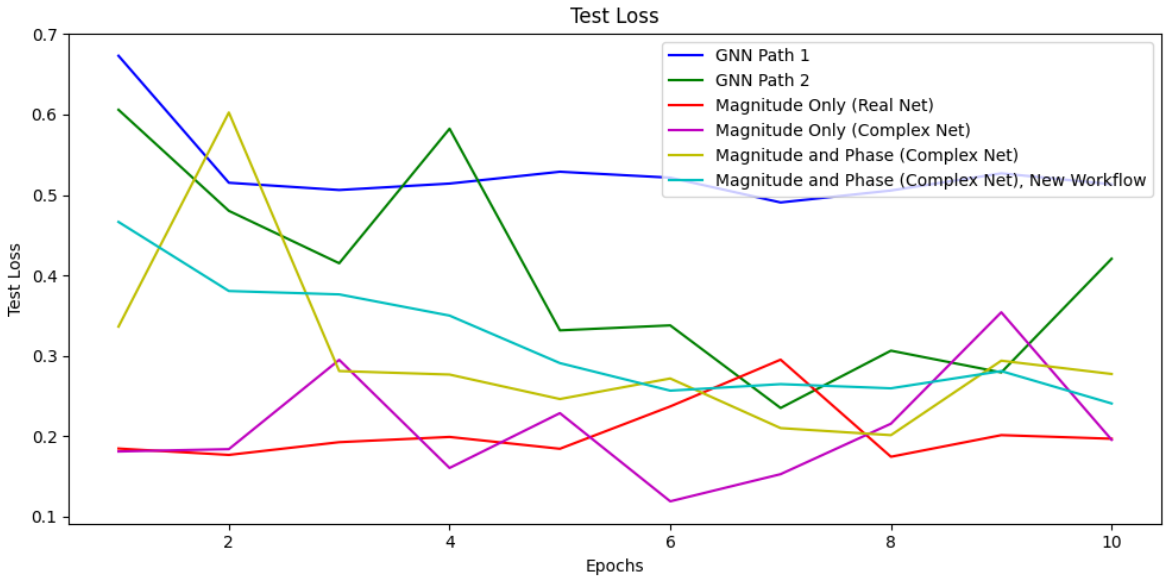
In [134…
```
# Data for the four scenarios
data = {
```

```
    "GNN Path 1": metrics_dict_e1,
    "GNN Path 2": metrics_dict_e2,
}
data.update(data_old)

# Data for plotting
epochs = range(1, 11)
colors = ['b', 'g', 'r', 'm', 'y', 'c', 'k', '#FF5733', '#7E4DFF']
scenarios = list(data.keys())

fig, axes = plt.subplots(2, 1, figsize=(10, 10))

for i, scenario in enumerate(scenarios):
    axes[0].plot(epochs, data[scenario]["train_accuracies"], label=scenar

axes[0].set_title("Train Accuracy")
axes[0].set_xlabel("Epochs")
axes[0].set_ylabel("Train Accuracy")
axes[0].legend()

for i, scenario in enumerate(scenarios):
    axes[1].plot(epochs, data[scenario]["test_accuracies"], label=scenari

axes[1].set_title("Test Accuracy")
axes[1].set_xlabel("Epochs")
axes[1].set_ylabel("Test Accuracy")
axes[1].legend()

plt.tight_layout()
plt.show()

fig, axes = plt.subplots(2, 1, figsize=(10, 10))

for i, scenario in enumerate(scenarios):
    axes[0].plot(epochs, data[scenario]["train_losses"], label=scenario,

axes[0].set_title("Train Loss")
axes[0].set_xlabel("Epochs")
axes[0].set_ylabel("Train Loss")
axes[0].legend()

for i, scenario in enumerate(scenarios):
    axes[1].plot(epochs, data[scenario]["test_losses"], label=scenario, c

axes[1].set_title("Test Loss")
axes[1].set_xlabel("Epochs")
axes[1].set_ylabel("Test Loss")
axes[1].legend()

plt.tight_layout()
plt.show()

fig, axes = plt.subplots(1, 1, figsize=(10, 5))
for i, scenario in enumerate(scenarios):
    axes.plot(epochs, data[scenario]["epoch_times"], label=scenario, colo
axes.set_title("Time")
axes.set_xlabel("Epochs")
axes.set_ylabel("Time (secs)")
axes.legend()
```


Train Accuracy
Train Accuracy
Epochs
GNN Path 1
GNN Path 2
Magnitude Only (Real Net)
Magnitude Only (Complex Net)
Magnitude and Phase (Complex Net)
Magnitude and Phase (Complex Net), New Workflow


Test Accuracy
Test Accuracy
Epochs
GNN Path 1
GNN Path 2
Magnitude Only (Real Net)
Magnitude Only (Complex Net)
Magnitude and Phase (Complex Net)
Magnitude and Phase (Complex Net), New Workflow





Out[134]: <matplotlib.legend.Legend at 0x2bba52d10>