Complex PyTorch for Music Genre Classification

In [143…

```
# 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 [144…

```
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 [145…

```
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 [146…

```
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 [147…

```
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 [148…

```
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 [149…

```
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 [150…

```
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, 2)

    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: 0.736654  Accuracy: 46.88%
Train Epoch:   0 [   320/  1680 ( 19%)] Loss: 0.510191  Accuracy: 57.10%
Train Epoch:   0 [   640/  1680 ( 38%)] Loss: 0.590632  Accuracy: 64.43%
Train Epoch:   0 [   960/  1680 ( 58%)] Loss: 0.523838  Accuracy: 66.53%
Train Epoch:   0 [  1280/  1680 ( 77%)] Loss: 0.549923  Accuracy: 68.60%
Train Epoch:   0 [  1600/  1680 ( 96%)] Loss: 0.400286  Accuracy: 70.83%
Epoch 0 - Time: 50.93s - Train Loss: 0.615360 - Train Accuracy: 70.24%
Test Loss: 0.524942 - Test Accuracy: 76.56%

Train Epoch:   1 [     0/  1680 (  0%)] Loss: 0.664110  Accuracy: 65.62%
Train Epoch:   1 [   320/  1680 ( 19%)] Loss: 0.464441  Accuracy: 71.88%
Train Epoch:   1 [   640/  1680 ( 38%)] Loss: 0.783113  Accuracy: 74.40%
Train Epoch:   1 [   960/  1680 ( 58%)] Loss: 0.413191  Accuracy: 75.20%
Train Epoch:   1 [  1280/  1680 ( 77%)] Loss: 0.892145  Accuracy: 75.53%
Train Epoch:   1 [  1600/  1680 ( 96%)] Loss: 0.495988  Accuracy: 74.94%
Epoch 1 - Time: 47.38s - Train Loss: 0.522190 - Train Accuracy: 74.11%
Test Loss: 0.545288 - Test Accuracy: 74.69%

Train Epoch:   2 [     0/  1680 (  0%)] Loss: 0.478801  Accuracy: 78.12%
Train Epoch:   2 [   320/  1680 ( 19%)] Loss: 0.334728  Accuracy: 79.26%
Train Epoch:   2 [   640/  1680 ( 38%)] Loss: 0.659334  Accuracy: 78.27%
Train Epoch:   2 [   960/  1680 ( 58%)] Loss: 0.576299  Accuracy: 76.51%
Train Epoch:   2 [  1280/  1680 ( 77%)] Loss: 0.495990  Accuracy: 76.37%
Train Epoch:   2 [  1600/  1680 ( 96%)] Loss: 0.521743  Accuracy: 75.86%
Epoch 2 - Time: 48.11s - Train Loss: 0.508656 - Train Accuracy: 75.24%
Test Loss: 0.533891 - Test Accuracy: 76.88%

Train Epoch:   3 [     0/  1680 (  0%)] Loss: 0.450253  Accuracy: 93.75%
Train Epoch:   3 [   320/  1680 ( 19%)] Loss: 0.646030  Accuracy: 79.83%
Train Epoch:   3 [   640/  1680 ( 38%)] Loss: 0.335803  Accuracy: 79.46%
Train Epoch:   3 [   960/  1680 ( 58%)] Loss: 0.516434  Accuracy: 78.33%
Train Epoch:   3 [  1280/  1680 ( 77%)] Loss: 0.476765  Accuracy: 77.13%
Train Epoch:   3 [  1600/  1680 ( 96%)] Loss: 0.385427  Accuracy: 76.90%
Epoch 3 - Time: 49.53s - Train Loss: 0.505611 - Train Accuracy: 76.19%
Test Loss: 0.526377 - Test Accuracy: 75.62%

Train Epoch:   4 [     0/  1680 (  0%)] Loss: 0.449090  Accuracy: 78.12%
Train Epoch:   4 [   320/  1680 ( 19%)] Loss: 0.548208  Accuracy: 75.85%
Train Epoch:   4 [   640/  1680 ( 38%)] Loss: 0.471726  Accuracy: 75.74%
Train Epoch:   4 [   960/  1680 ( 58%)] Loss: 0.413275  Accuracy: 77.32%
Train Epoch:   4 [  1280/  1680 ( 77%)] Loss: 0.369933  Accuracy: 76.75%
Train Epoch:   4 [  1600/  1680 ( 96%)] Loss: 0.489496  Accuracy: 76.84%
Epoch 4 - Time: 48.32s - Train Loss: 0.500867 - Train Accuracy: 75.65%
Test Loss: 0.516914 - Test Accuracy: 75.94%

Train Epoch:   5 [     0/  1680 (  0%)] Loss: 0.516894  Accuracy: 71.88%
Train Epoch:   5 [   320/  1680 ( 19%)] Loss: 0.477135  Accuracy: 75.85%
Train Epoch:   5 [   640/  1680 ( 38%)] Loss: 0.436307  Accuracy: 75.74%
Train Epoch:   5 [   960/  1680 ( 58%)] Loss: 0.602909  Accuracy: 75.30%
Train Epoch:   5 [  1280/  1680 ( 77%)] Loss: 0.604455  Accuracy: 76.07%
Train Epoch:   5 [  1600/  1680 ( 96%)] Loss: 0.386279  Accuracy: 76.16%
Epoch 5 - Time: 47.54s - Train Loss: 0.505030 - Train Accuracy: 75.65%
Test Loss: 0.505044 - Test Accuracy: 77.19%

Train Epoch:   6 [     0/  1680 (  0%)] Loss: 0.445658  Accuracy: 81.25%
Train Epoch:   6 [   320/  1680 ( 19%)] Loss: 0.526011  Accuracy: 75.57%
Train Epoch:   6 [   640/  1680 ( 38%)] Loss: 0.504155  Accuracy: 75.15%
Train Epoch:   6 [   960/  1680 ( 58%)] Loss: 0.418555  Accuracy: 77.02%
Train Epoch:   6 [  1280/  1680 ( 77%)] Loss: 0.464673  Accuracy: 78.28%
Train Epoch:   6 [  1600/  1680 ( 96%)] Loss: 0.482963  Accuracy: 78.00%
```

```
Epoch 6 - Time: 47.54s - Train Loss: 0.487412 - Train Accuracy: 77.20%
Test Loss: 0.597018 - Test Accuracy: 72.81%

Train Epoch:   7 [     0/  1680 (  0%)] Loss: 0.487195  Accuracy: 75.00%
Train Epoch:   7 [   320/  1680 ( 19%)] Loss: 0.329567  Accuracy: 79.83%
Train Epoch:   7 [   640/  1680 ( 38%)] Loss: 0.463091  Accuracy: 79.02%
Train Epoch:   7 [   960/  1680 ( 58%)] Loss: 0.569358  Accuracy: 77.52%
Train Epoch:   7 [  1280/  1680 ( 77%)] Loss: 0.431351  Accuracy: 77.21%
Train Epoch:   7 [  1600/  1680 ( 96%)] Loss: 0.495203  Accuracy: 76.53%
Epoch 7 - Time: 51.83s - Train Loss: 0.500928 - Train Accuracy: 75.83%
Test Loss: 0.559238 - Test Accuracy: 74.38%

Train Epoch:   8 [     0/  1680 (  0%)] Loss: 0.759799  Accuracy: 59.38%
Train Epoch:   8 [   320/  1680 ( 19%)] Loss: 0.708896  Accuracy: 72.44%
Train Epoch:   8 [   640/  1680 ( 38%)] Loss: 0.650355  Accuracy: 75.30%
Train Epoch:   8 [   960/  1680 ( 58%)] Loss: 0.373965  Accuracy: 77.22%
Train Epoch:   8 [  1280/  1680 ( 77%)] Loss: 0.406738  Accuracy: 78.51%
Train Epoch:   8 [  1600/  1680 ( 96%)] Loss: 0.596653  Accuracy: 78.74%
Epoch 8 - Time: 56.79s - Train Loss: 0.470946 - Train Accuracy: 78.15%
Test Loss: 0.585353 - Test Accuracy: 77.81%

Train Epoch:   9 [     0/  1680 (  0%)] Loss: 0.390343  Accuracy: 87.50%
Train Epoch:   9 [   320/  1680 ( 19%)] Loss: 0.311492  Accuracy: 81.82%
Train Epoch:   9 [   640/  1680 ( 38%)] Loss: 0.738014  Accuracy: 77.23%
Train Epoch:   9 [   960/  1680 ( 58%)] Loss: 0.508481  Accuracy: 77.42%
Train Epoch:   9 [  1280/  1680 ( 77%)] Loss: 0.550660  Accuracy: 77.29%
Train Epoch:   9 [  1600/  1680 ( 96%)] Loss: 0.548808  Accuracy: 76.78%
Epoch 9 - Time: 50.51s - Train Loss: 0.499331 - Train Accuracy: 75.89%
Test Loss: 0.531108 - Test Accuracy: 75.94%

---------------------------------------------------------------------------
--------------------------
---------------------------------------------------------------------------
--------------------------
FINAL RESULTS:
---------------------------------------------------------------------------
--------------------------
epoch_times: [50.92506289482117, 47.37869906425476, 48.109081745147705, 4
9.534204959869385, 48.31972289085388, 47.53947877883911, 47.53571987152099
6, 51.829482078552246, 56.78982973098755, 50.50758194923401]
train_losses: [0.6153599700102439, 0.5221903874323919, 0.5086564530546849,
0.5056113818517098, 0.5008668750524521, 0.505030136841994, 0.4874119466313
949, 0.5009277480152937, 0.47094558294002825, 0.49933138489723206]
train_accuracies: [70.23809523809524, 74.10714285714286, 75.2380952380952
4, 76.19047619047619, 75.6547619047619, 75.6547619047619, 77.2023809523809
5, 75.83333333333333, 78.1547619047619, 75.89285714285714]
test_losses: [0.5249415069818497, 0.5452884390950203, 0.533890588581562,
0.526377010345459, 0.5169135600328445, 0.5050437748432159, 0.5970180429518
223, 0.559237914532423, 0.5853527415543794, 0.5311079621315002]
test_accuracies: [76.5625, 74.6875, 76.875, 75.625, 75.9375, 77.1875, 72.8
125, 74.375, 77.8125, 75.9375]
```

2. Simple Graph Net (Magniude + phase wieghts)

In [151…

```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 [152…

```
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, 2)

    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: 0.695338  Accuracy: 56.25%
Train Epoch:   0 [   320/  1680 ( 19%)] Loss: 2.676930  Accuracy: 59.94%
Train Epoch:   0 [   640/  1680 ( 38%)] Loss: 0.504754  Accuracy: 62.80%
Train Epoch:   0 [   960/  1680 ( 58%)] Loss: 1.063907  Accuracy: 66.63%
Train Epoch:   0 [  1280/  1680 ( 77%)] Loss: 0.498953  Accuracy: 70.05%
Train Epoch:   0 [  1600/  1680 ( 96%)] Loss: 0.215690  Accuracy: 73.22%
Epoch 0 - Time: 47.69s - Train Loss: 0.713414 - Train Accuracy: 72.38%
Test Loss: 0.473514 - Test Accuracy: 77.19%

Train Epoch:   1 [     0/  1680 (  0%)] Loss: 0.259072  Accuracy: 90.62%
Train Epoch:   1 [   320/  1680 ( 19%)] Loss: 0.585487  Accuracy: 82.95%
Train Epoch:   1 [   640/  1680 ( 38%)] Loss: 0.256859  Accuracy: 83.78%
Train Epoch:   1 [   960/  1680 ( 58%)] Loss: 0.211788  Accuracy: 84.07%
Train Epoch:   1 [  1280/  1680 ( 77%)] Loss: 0.271385  Accuracy: 84.60%
Train Epoch:   1 [  1600/  1680 ( 96%)] Loss: 0.221916  Accuracy: 84.99%
Epoch 1 - Time: 47.76s - Train Loss: 0.362272 - Train Accuracy: 84.23%
Test Loss: 0.352983 - Test Accuracy: 85.62%

Train Epoch:   2 [     0/  1680 (  0%)] Loss: 0.245913  Accuracy: 87.50%
Train Epoch:   2 [   320/  1680 ( 19%)] Loss: 0.553280  Accuracy: 82.39%
Train Epoch:   2 [   640/  1680 ( 38%)] Loss: 0.299612  Accuracy: 84.23%
Train Epoch:   2 [   960/  1680 ( 58%)] Loss: 0.309136  Accuracy: 84.27%
Train Epoch:   2 [  1280/  1680 ( 77%)] Loss: 0.307497  Accuracy: 84.98%
Train Epoch:   2 [  1600/  1680 ( 96%)] Loss: 0.364019  Accuracy: 84.25%
Epoch 2 - Time: 51.44s - Train Loss: 0.371450 - Train Accuracy: 83.57%
Test Loss: 0.310048 - Test Accuracy: 90.31%

Train Epoch:   3 [     0/  1680 (  0%)] Loss: 0.369656  Accuracy: 81.25%
Train Epoch:   3 [   320/  1680 ( 19%)] Loss: 0.310413  Accuracy: 86.93%
Train Epoch:   3 [   640/  1680 ( 38%)] Loss: 0.247773  Accuracy: 87.80%
Train Epoch:   3 [   960/  1680 ( 58%)] Loss: 0.383761  Accuracy: 85.58%
Train Epoch:   3 [  1280/  1680 ( 77%)] Loss: 0.508220  Accuracy: 85.29%
Train Epoch:   3 [  1600/  1680 ( 96%)] Loss: 0.434038  Accuracy: 84.99%
Epoch 3 - Time: 52.10s - Train Loss: 0.346548 - Train Accuracy: 84.29%
Test Loss: 0.344095 - Test Accuracy: 84.69%

Train Epoch:   4 [     0/  1680 (  0%)] Loss: 0.371787  Accuracy: 84.38%
Train Epoch:   4 [   320/  1680 ( 19%)] Loss: 0.309196  Accuracy: 85.23%
Train Epoch:   4 [   640/  1680 ( 38%)] Loss: 0.392832  Accuracy: 87.05%
Train Epoch:   4 [   960/  1680 ( 58%)] Loss: 0.240444  Accuracy: 86.90%
Train Epoch:   4 [  1280/  1680 ( 77%)] Loss: 0.290675  Accuracy: 86.89%
Train Epoch:   4 [  1600/  1680 ( 96%)] Loss: 0.312086  Accuracy: 86.83%
Epoch 4 - Time: 52.38s - Train Loss: 0.319339 - Train Accuracy: 86.01%
Test Loss: 0.379172 - Test Accuracy: 83.44%

Train Epoch:   5 [     0/  1680 (  0%)] Loss: 0.634693  Accuracy: 81.25%
Train Epoch:   5 [   320/  1680 ( 19%)] Loss: 0.401959  Accuracy: 85.23%
Train Epoch:   5 [   640/  1680 ( 38%)] Loss: 0.307199  Accuracy: 85.12%
Train Epoch:   5 [   960/  1680 ( 58%)] Loss: 0.401353  Accuracy: 84.98%
Train Epoch:   5 [  1280/  1680 ( 77%)] Loss: 0.313377  Accuracy: 86.28%
Train Epoch:   5 [  1600/  1680 ( 96%)] Loss: 0.367686  Accuracy: 86.83%
Epoch 5 - Time: 52.31s - Train Loss: 0.322532 - Train Accuracy: 85.83%
Test Loss: 0.267910 - Test Accuracy: 90.94%

Train Epoch:   6 [     0/  1680 (  0%)] Loss: 0.141243  Accuracy: 96.88%
Train Epoch:   6 [   320/  1680 ( 19%)] Loss: 0.444822  Accuracy: 89.49%
Train Epoch:   6 [   640/  1680 ( 38%)] Loss: 0.441684  Accuracy: 90.03%
Train Epoch:   6 [   960/  1680 ( 58%)] Loss: 0.239080  Accuracy: 90.02%
Train Epoch:   6 [  1280/  1680 ( 77%)] Loss: 0.224162  Accuracy: 90.40%
Train Epoch:   6 [  1600/  1680 ( 96%)] Loss: 0.453201  Accuracy: 89.34%
```

```
Epoch 6 - Time: 48.56s - Train Loss: 0.282306 - Train Accuracy: 88.10%
Test Loss: 0.289120 - Test Accuracy: 93.12%

Train Epoch:   7 [     0/  1680 (  0%)] Loss: 0.197175  Accuracy: 93.75%
Train Epoch:   7 [   320/  1680 ( 19%)] Loss: 0.368956  Accuracy: 87.78%
Train Epoch:   7 [   640/  1680 ( 38%)] Loss: 0.176845  Accuracy: 88.39%
Train Epoch:   7 [   960/  1680 ( 58%)] Loss: 0.294597  Accuracy: 87.80%
Train Epoch:   7 [  1280/  1680 ( 77%)] Loss: 0.378605  Accuracy: 87.42%
Train Epoch:   7 [  1600/  1680 ( 96%)] Loss: 0.356275  Accuracy: 87.32%
Epoch 7 - Time: 48.90s - Train Loss: 0.304407 - Train Accuracy: 86.37%
Test Loss: 0.412482 - Test Accuracy: 84.38%

Train Epoch:   8 [     0/  1680 (  0%)] Loss: 0.437261  Accuracy: 84.38%
Train Epoch:   8 [   320/  1680 ( 19%)] Loss: 0.214728  Accuracy: 87.50%
Train Epoch:   8 [   640/  1680 ( 38%)] Loss: 0.262306  Accuracy: 87.35%
Train Epoch:   8 [   960/  1680 ( 58%)] Loss: 0.242323  Accuracy: 88.21%
Train Epoch:   8 [  1280/  1680 ( 77%)] Loss: 0.251799  Accuracy: 87.80%
Train Epoch:   8 [  1600/  1680 ( 96%)] Loss: 0.180843  Accuracy: 88.05%
Epoch 8 - Time: 49.62s - Train Loss: 0.302493 - Train Accuracy: 87.08%
Test Loss: 0.281022 - Test Accuracy: 88.12%

Train Epoch:   9 [     0/  1680 (  0%)] Loss: 0.225743  Accuracy: 90.62%
Train Epoch:   9 [   320/  1680 ( 19%)] Loss: 0.298144  Accuracy: 89.77%
Train Epoch:   9 [   640/  1680 ( 38%)] Loss: 0.392851  Accuracy: 88.84%
Train Epoch:   9 [   960/  1680 ( 58%)] Loss: 0.244691  Accuracy: 88.00%
Train Epoch:   9 [  1280/  1680 ( 77%)] Loss: 0.431879  Accuracy: 87.88%
Train Epoch:   9 [  1600/  1680 ( 96%)] Loss: 0.274541  Accuracy: 87.62%
Epoch 9 - Time: 51.28s - Train Loss: 0.303119 - Train Accuracy: 86.79%
Test Loss: 0.275992 - Test Accuracy: 92.50%

--------------------------------------------------------------------------
--------------------------
--------------------------------------------------------------------------
--------------------------
FINAL RESULTS:
--------------------------------------------------------------------------
--------------------------
epoch_times: [47.690855979919434, 47.75747728347778, 51.43562912940979, 5
2.09551191329956, 52.38199305534363, 52.30853605270386, 48.55864787101745
6, 48.90050506591797, 49.619580030441284, 51.27541995048523]
train_losses: [0.7134135881295571, 0.3622721763184437, 0.3714496562114128
5, 0.3465476266753215, 0.31933862902224064, 0.32253212424424976, 0.2823056
9798212785, 0.30440737235431486, 0.3024934452886765, 0.30311915937524575]
train_accuracies: [72.38095238095238, 84.22619047619048, 83.5714285714285
7, 84.28571428571429, 86.01190476190476, 85.83333333333333, 88.09523809523
81, 86.36904761904762, 87.08333333333333, 86.78571428571429]
test_losses: [0.47351393606513736, 0.35298285372555255, 0.310048015415668
5, 0.344094629958272, 0.3791716232895851, 0.2679101286455989, 0.2891197871
4168074, 0.41248247046023606, 0.28102156994864347, 0.27599221765995025]
test_accuracies: [77.1875, 85.625, 90.3125, 84.6875, 83.4375, 90.9375, 93.
125, 84.375, 88.125, 92.5]
```

Plots

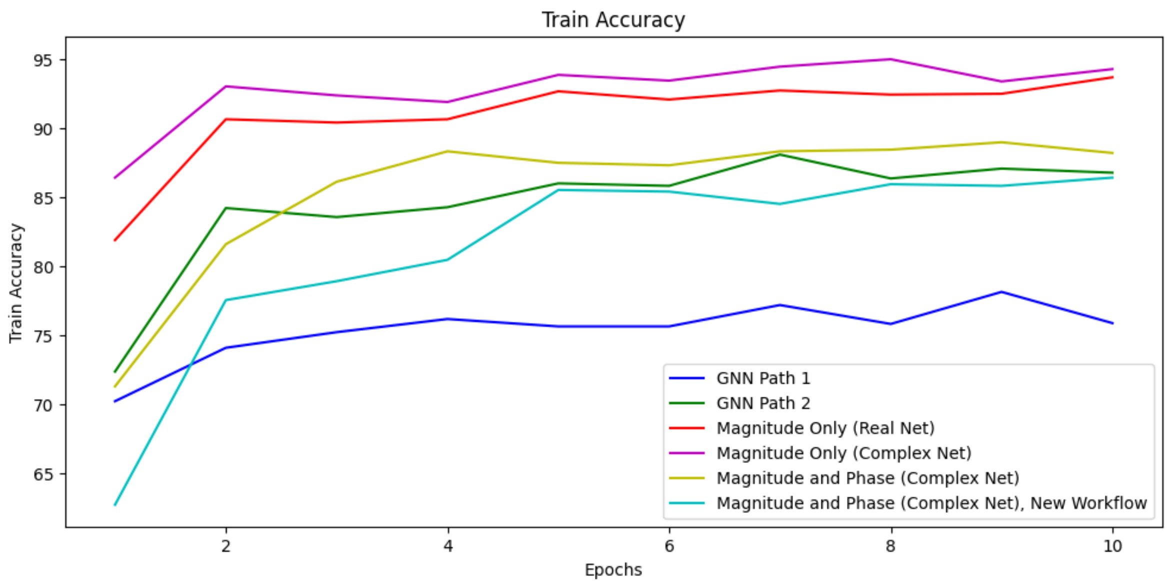
In [153…

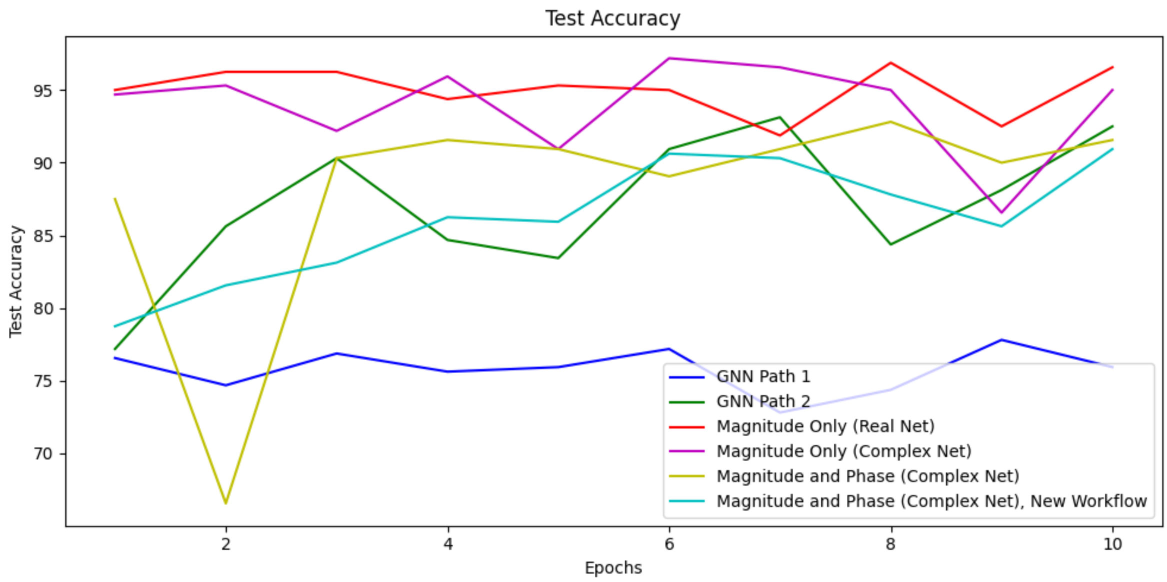
```python
data_old = {'Magnitude Only (Real Net)': {'epoch_times': [37.517344951629
```

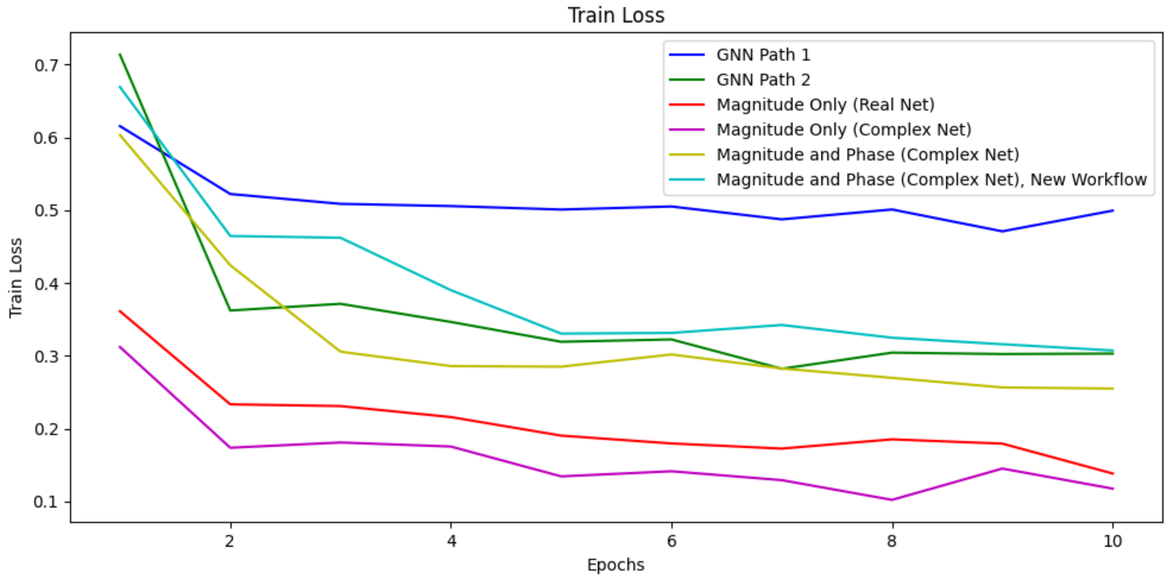
In [154…

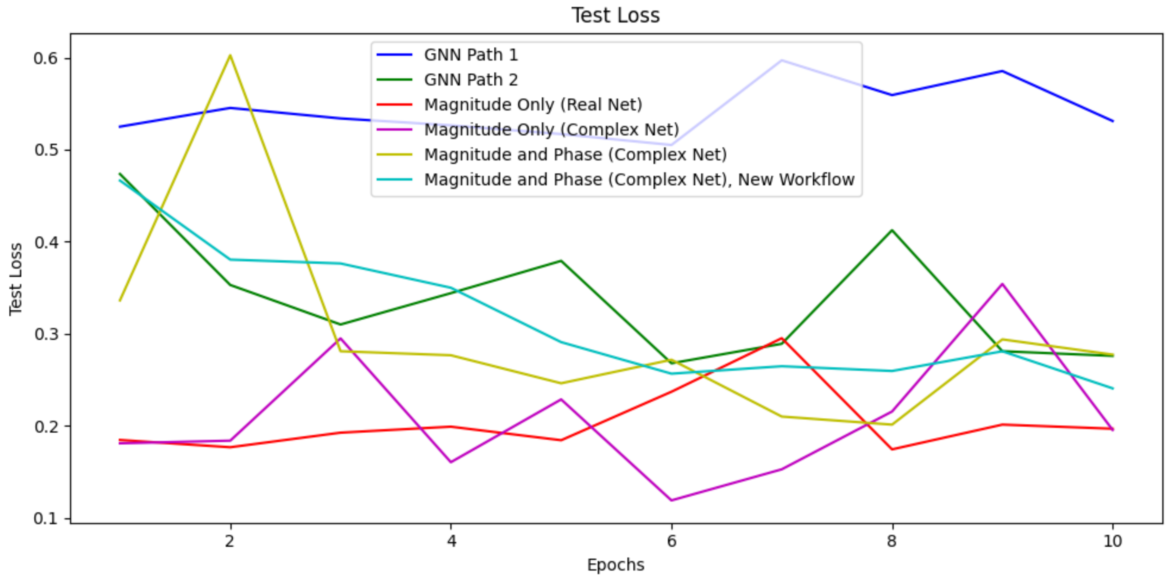
```python
# Data for the four scenarios
data = {
```

```
    "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
95
90
85
80
75
70
65
2
4
6
8
10
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
95
90
85
80
75
70
2
4
6
8
10
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[154]: <matplotlib.legend.Legend at 0x2bcfc94d0>