gudhi_cnn_combined

November 23, 2024

[16]:
```python
import numpy as np
import torch
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from tqdm.notebook import tqdm
from torchsummary import summary
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score, recall_score, f1_score
import pandas as pd
```

[17]:
```python
# import labels for the shapes
shape_labels = np.genfromtxt('Gudhi Shape Dataset/shape_labels.csv',␣
↪delimiter=',', skip_header=1)
shape_labels = shape_labels.astype(int)[:,2]
print(shape_labels)
```

```
[1 1 1 … 0 0 0]
```

[ ]:
```python
num_samples = 2000 # currently set to full dataset

# Generate random indices
random_indices = np.random.choice(len(shape_labels), size=num_samples,␣
↪replace=False)

# Select the corresponding data and labels
laplacians = []
vr_persistence_images = []
abstract_persistence_images = []
selected_labels = []

for i in random_indices:
    laplacians.append(np.genfromtxt(f'Gudhi Shape Dataset/shape_{i}_laplacian.
↪csv', delimiter=',', skip_header=0))
    vr_persistence_images.append(np.genfromtxt(f'Gudhi Shape Dataset/
↪shape_{i}_vr_persistence_image.csv', delimiter=',', skip_header=0))
```

```
    abstract_persistence_images.append(np.genfromtxt(f'Gudhi Shape Dataset/
↪shape_{i}_abstract_persistence_image.csv', delimiter=',', skip_header=0))
    selected_labels.append(shape_labels[i])

# Convert selected labels to NumPy array
selected_labels = np.array(selected_labels)

# Print a summary
print(f"Randomly selected {num_samples} samples.")
print(f"Shape of laplacians: {np.array(laplacians).shape}")
print(f"Shape of VR persistence images: {np.array(vr_persistence_images).
↪shape}")
print(f"Shape of abstract persistence images: {np.
↪array(abstract_persistence_images).shape}")
print(f"Shape of selected labels: {selected_labels.shape}")
```

```
Randomly selected 2000 samples.
Shape of laplacians: (2000, 1000, 1000)
Shape of VR persistence images: (2000, 100, 100)
Shape of abstract persistence images: (2000, 100, 100)
Shape of selected labels: (2000,)
```

[ ]:
```
class ShapeDataset(torch.utils.data.Dataset):
    def __init__(self, data, labels):
        self.data = [torch.tensor(d, dtype=torch.float32).unsqueeze(0) for d in␣
↪data]
        self.labels = torch.tensor(labels, dtype=torch.long)

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

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

[20]:
```
class CNN(nn.Module):
    def __init__(self, input_shape, num_classes=2):
        super(CNN, self).__init__()
        # Convolutional Layers
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)

        # Pooling Layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Adaptive Pooling to resize to 100x100
        self.adaptive_pool = nn.AdaptiveAvgPool2d((100, 100))
```

```
        # Dynamically calculate input size to fc1
        self.feature_size = self._get_feature_size(input_shape)

        # Fully Connected Layers
        self.fc1 = nn.Linear(self.feature_size, 128)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(128, num_classes)

    def _get_feature_size(self, input_shape):
        # Create a dummy input to calculate size after conv and pooling
        dummy_input = torch.zeros(1, 1, *input_shape)
        x = self.pool(F.relu(self.conv1(dummy_input)))
        x = self.pool(F.relu(self.conv2(x)))

        # Apply adaptive pooling to get 100x100 size
        x = self.adaptive_pool(x)
        return x.numel()  # Number of elements after flattening

    def forward(self, x):
        # Apply convolutional layers with pooling
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))

        # Apply adaptive pooling to resize to 100x100
        x = self.adaptive_pool(x)

        # Flatten and pass through fully connected layers
        x = torch.flatten(x, start_dim=1)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

[21]:
```
class DualInputCNN(nn.Module):
    def __init__(self, input_shape1, input_shape2, num_classes=2):
        super(DualInputCNN, self).__init__()

        # Laplacian input path with additional pooling to reduce to 100x100
        self.conv1_lap = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        self.conv2_lap = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.pool_lap = nn.MaxPool2d(2, 2)  # Reduce spatial dimensions
        self.adaptive_pool_lap = nn.AdaptiveAvgPool2d((100, 100))  # Resize to
 ↪100x100

        # Persistence image input path (no pooling)
        self.conv1_pers = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        self.conv2_pers = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
```

```
        self.adaptive_pool_pers = nn.AdaptiveAvgPool2d((100, 100))  # Resize to␣
↪100x100

        # Fully connected layers
        self.fc1 = nn.Linear(32 * 100 * 100 + 32 * 100 * 100, 128)  # Adjusted␣
↪for 100x100 input
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(128, num_classes)

    def forward(self, x1, x2):
        # Laplacians path (downsampling to 100x100)
        x1 = F.relu(self.conv1_lap(x1))
        x1 = self.pool_lap(x1)  # First pool: 250x250 -> 125x125
        x1 = F.relu(self.conv2_lap(x1))
        x1 = self.pool_lap(x1)  # Second pool: 125x125 -> 62x62
        x1 = self.adaptive_pool_lap(x1)  # Resize to 100x100

        # Persistence images path (no pooling)
        x2 = F.relu(self.conv1_pers(x2))
        x2 = F.relu(self.conv2_pers(x2))
        x2 = self.adaptive_pool_pers(x2)  # Ensure persistence images are␣
↪100x100

        # Concatenate along dim=1 (channels)
        x = torch.cat((x1, x2), dim=1)  # Concatenates the outputs along the␣
↪channel axis

        # Flatten for fully connected layer
        x = torch.flatten(x, start_dim=1)

        # Fully connected layers
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

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

[22]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Training on {device}")
```

```
Training on cpu
```

[23]:
```
def train_and_test_single_input(data_type, data, labels, input_shape):
    # Create dataset and split into train/test sets
    dataset = ShapeDataset(data, labels)
    train_data, test_data, train_labels, test_labels = train_test_split(
        dataset.data, dataset.labels, test_size=0.2, random_state=42
```

```
    )

    # Convert to custom Dataset format for train and test sets
    train_dataset = torch.utils.data.TensorDataset(torch.stack(train_data),␣
↪train_labels)
    test_dataset = torch.utils.data.TensorDataset(torch.stack(test_data),␣
↪test_labels)

    # Create DataLoaders
    train_dataloader = torch.utils.data.DataLoader(train_dataset,␣
↪batch_size=16, shuffle=True)
    test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=32,␣
↪shuffle=False)

    # Define CNN model with input_shape
    model = CNN(input_shape=input_shape, num_classes=len(set(labels))).
↪to(device)

    # Define optimizer and loss function
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    criterion = nn.CrossEntropyLoss()  # For multi-class classification

    epoch_results = []

    # Training loop
    num_epochs = 10
    for epoch in range(1, num_epochs + 1):
        print(f"Training model for {data_type} - Epoch {epoch}/{num_epochs}")
        train_loss, accuracy = train_single_input(model, device,␣
↪train_dataloader, optimizer, criterion, epoch)
        test_loss, accuracy, precision, recall, f1 = test_single_input(model,␣
↪device, test_dataloader, criterion)

        # Store results
        epoch_results.append({
            'Epoch': epoch,
            'Test Loss': test_loss,
            'Test Accuracy (%)': accuracy,
            'Test Precision (%)': precision * 100,
            'Test Recall (%)': recall * 100,
            'Test F1 Score': f1
        })

    # Convert results to DataFrame for tabular output
    epoch_results_df = pd.DataFrame(epoch_results)
    print(f"\nTesting results for {data_type}:")
```

```
    print(epoch_results_df)
```

```
[ ]: def train_and_test_dual_input(data_type, data1, data2, labels, input_shape1,␣
↪input_shape2):
    dataset1 = [torch.tensor(d, dtype=torch.float32).unsqueeze(0) for d in␣
↪data1]  # Shape [1, 100, 100]
    dataset2 = [torch.tensor(d, dtype=torch.float32).unsqueeze(0) for d in␣
↪data2]  # Shape [1, 100, 100]
    labels = torch.tensor(labels, dtype=torch.long)

    train_data1, test_data1, train_data2, test_data2, train_labels, test_labels␣
↪= train_test_split(
        dataset1, dataset2, labels, test_size=0.2, random_state=42
    )

    train_dataset = torch.utils.data.TensorDataset(
        torch.stack(train_data1), torch.stack(train_data2), train_labels
    )
    test_dataset = torch.utils.data.TensorDataset(
        torch.stack(test_data1), torch.stack(test_data2), test_labels
    )

    train_dataloader = torch.utils.data.DataLoader(train_dataset,␣
↪batch_size=16, shuffle=True)
    test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=32,␣
↪shuffle=False)

    model = DualInputCNN(input_shape1=input_shape1, input_shape2=input_shape2,␣
↪num_classes=len(set(labels))).to(device)

    optimizer = optim.Adam(model.parameters(), lr=0.001)
    criterion = nn.CrossEntropyLoss()

    epoch_results = []

    num_epochs = 10
    for epoch in range(1, num_epochs + 1):
        print(f"Training model for {data_type} - Epoch {epoch}/5")

        # Training step
        train_loss, train_accuracy = train_dual_input(model, device,␣
↪train_dataloader, optimizer, criterion, epoch)

        # Testing step
        test_loss, test_accuracy, precision, recall, f1_score =␣
↪test_dual_input(model, device, test_dataloader, criterion)
```

```
        # Storing results
        epoch_results.append({
            'Epoch': epoch,
            'Test Loss': test_loss,
            'Test Accuracy (%)': test_accuracy,
            'Precision': precision,
            'Recall': recall,
            'F1 Score': f1_score
        })

    # Convert results to DataFrame for tabular output
    epoch_results_df = pd.DataFrame(epoch_results)
    print(f"\nTesting results for {data_type}:")
    print(epoch_results_df)
```

[26]:

```
def train_single_input(model, device, train_loader, optimizer, criterion,␣
 ↪epoch):
    model.train()
    train_loss = 0
    correct = 0
    total = 0
    tk0 = tqdm(train_loader, total=len(train_loader))
    for batch_idx, (data, target) in enumerate(tk0):
        data, target = data.to(device), target.to(device)

        # Zero gradients
        optimizer.zero_grad()

        # Forward pass with single input
        output = model(data)  # Forward through the single-input model

        # Compute loss
        loss = criterion(output, target)

        # Backward pass and optimize
        loss.backward()
        optimizer.step()

        # Update loss and accuracy
        train_loss += loss.item()
        _, predicted = output.max(1)
        total += target.size(0)
        correct += predicted.eq(target).sum().item()

        # Update progress bar
        tk0.set_postfix(loss=loss.item())
```

```
    avg_loss = train_loss / len(train_loader)
    accuracy = 100. * correct / total
    return avg_loss, accuracy


def test_single_input(model, device, test_loader, criterion):
    model.eval()
    test_loss = 0
    correct = 0
    total = 0
    all_preds = []
    all_targets = []

    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)  # Forward through the single-input model
            loss = criterion(output, target)

            test_loss += loss.item()
            _, predicted = output.max(1)
            total += target.size(0)
            correct += predicted.eq(target).sum().item()

            all_preds.extend(predicted.cpu().numpy())
            all_targets.extend(target.cpu().numpy())

    avg_loss = test_loss / len(test_loader)
    accuracy = 100. * correct / total

    # Calculate additional metrics
    precision = precision_score(all_targets, all_preds, average='weighted')
    recall = recall_score(all_targets, all_preds, average='weighted')
    f1 = f1_score(all_targets, all_preds, average='weighted')

    return avg_loss, accuracy, precision, recall, f1
```

[27]:

```
def train_dual_input(model, device, train_loader, optimizer, criterion, epoch):
    model.train()
    train_loss = 0
    correct = 0
    total = 0
    tk0 = tqdm(train_loader, total=len(train_loader))
    for batch_idx, (data1, data2, target) in enumerate(tk0):
        data1, data2, target = data1.to(device), data2.to(device), target.
↪to(device)
```

```
        # Zero gradients
        optimizer.zero_grad()

        # Forward pass with dual input
        output = model(data1, data2)  # Forward through the dual-input model

        # Compute loss
        loss = criterion(output, target)

        # Backward pass and optimize
        loss.backward()
        optimizer.step()

        # Update loss and accuracy
        train_loss += loss.item()
        _, predicted = output.max(1)
        total += target.size(0)
        correct += predicted.eq(target).sum().item()

        # Update progress bar
        tk0.set_postfix(loss=loss.item())

    avg_loss = train_loss / len(train_loader)
    accuracy = 100. * correct / total
    return avg_loss, accuracy


def test_dual_input(model, device, test_loader, criterion):
    model.eval()
    test_loss = 0
    correct = 0
    total = 0
    all_labels = []
    all_preds = []

    with torch.no_grad():
        for data1, data2, target in test_loader:
            data1, data2, target = data1.to(device), data2.to(device), target.
↪to(device)
            output = model(data1, data2)  # Forward through the dual-input model
            loss = criterion(output, target)

            test_loss += loss.item()
            _, predicted = output.max(1)
            total += target.size(0)
            correct += predicted.eq(target).sum().item()
```

```
            # Collect all labels and predictions for additional metrics
            all_labels.extend(target.cpu().numpy())
            all_preds.extend(predicted.cpu().numpy())

    avg_loss = test_loss / len(test_loader)
    accuracy = 100. * correct / total

    # Calculate additional metrics
    precision = precision_score(all_labels, all_preds, average='weighted')
    recall = recall_score(all_labels, all_preds, average='weighted')
    f1 = f1_score(all_labels, all_preds, average='weighted')

    # Return test loss, accuracy, and additional metrics
    return avg_loss, accuracy, precision, recall, f1
```

[28]:
```
# Train and test for Laplacians (1000x1000 input)
train_and_test_single_input("Laplacians", laplacians, selected_labels,␣
 ↪input_shape=(1000, 1000))
```

```
Training model for Laplacians - Epoch 1/10

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians - Epoch 2/10

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians - Epoch 3/10

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians - Epoch 4/10

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians - Epoch 5/10

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians - Epoch 6/10

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians - Epoch 7/10

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians - Epoch 8/10

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians - Epoch 9/10

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians - Epoch 10/10
```

```
  0%|          | 0/100 [00:00<?, ?it/s]

Testing results for Laplacians:
   Epoch  Test Loss  Test Accuracy (%)  Test Precision (%)  Test Recall (%)  \
0      1   0.162553              95.00           95.427350            95.00
1      2   0.158385              95.50           95.849138            95.50
2      3   0.173197              95.00           95.427350            95.00
3      4   0.124448              96.00           96.209914            96.00
4      5   0.177778              95.25           95.556021            95.25
5      6   0.214507              92.75           92.784992            92.75
6      7   0.131869              96.00           96.151358            96.00
7      8   0.141956              96.75           96.757158            96.75
8      9   0.161437              95.50           95.849138            95.50
9     10   0.193524              94.50           94.593344            94.50

   Test F1 Score
0       0.949696
1       0.954764
2       0.949696
3       0.959851
4       0.952270
5       0.927388
6       0.959879
7       0.967481
8       0.954764
9       0.944868
```

[29]:
```
train_and_test_dual_input("Laplacians + VR Persistence Images", laplacians,
  vr_persistence_images, selected_labels, (1000, 1000), (100, 100))
```

```
Training model for Laplacians + VR Persistence Images - Epoch 1/5

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians + VR Persistence Images - Epoch 2/5

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians + VR Persistence Images - Epoch 3/5

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians + VR Persistence Images - Epoch 4/5

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians + VR Persistence Images - Epoch 5/5

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians + VR Persistence Images - Epoch 6/5

  0%|          | 0/100 [00:00<?, ?it/s]
```

```
Training model for Laplacians + VR Persistence Images - Epoch 7/5

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians + VR Persistence Images - Epoch 8/5

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians + VR Persistence Images - Epoch 9/5

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians + VR Persistence Images - Epoch 10/5

  0%|          | 0/100 [00:00<?, ?it/s]


Testing results for Laplacians + VR Persistence Images:
   Epoch  Test Loss  Test Accuracy (%)  Precision  Recall  F1 Score
0      1   0.028382              99.75   0.997512  0.9975  0.997500
1      2   0.206494              98.50   0.985469  0.9850  0.985012
2      3   0.002265              99.75   0.997512  0.9975  0.997500
3      4   0.062603              99.00   0.990211  0.9900  0.990006
4      5   0.040268              99.25   0.992604  0.9925  0.992496
5      6   0.030000              99.25   0.992604  0.9925  0.992496
6      7   0.000764             100.00   1.000000  1.0000  1.000000
7      8   0.000685             100.00   1.000000  1.0000  1.000000
8      9   0.000312             100.00   1.000000  1.0000  1.000000
9     10   0.016074              99.75   0.997512  0.9975  0.997500
```

[30]:
```python
train_and_test_dual_input("Laplacians + Abstract Persistence Images",
 ↪laplacians, abstract_persistence_images, selected_labels, (1000, 1000),
 ↪(100, 100))
```

```
Training model for Laplacians + Abstract Persistence Images - Epoch 1/5

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians + Abstract Persistence Images - Epoch 2/5

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians + Abstract Persistence Images - Epoch 3/5

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians + Abstract Persistence Images - Epoch 4/5

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians + Abstract Persistence Images - Epoch 5/5

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians + Abstract Persistence Images - Epoch 6/5

  0%|          | 0/100 [00:00<?, ?it/s]
```

```
Training model for Laplacians + Abstract Persistence Images - Epoch 7/5

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians + Abstract Persistence Images - Epoch 8/5

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians + Abstract Persistence Images - Epoch 9/5

  0%|          | 0/100 [00:00<?, ?it/s]

Training model for Laplacians + Abstract Persistence Images - Epoch 10/5

  0%|          | 0/100 [00:00<?, ?it/s]


Testing results for Laplacians + Abstract Persistence Images:
   Epoch  Test Loss  Test Accuracy (%)  Precision  Recall  F1 Score
0      1   0.060849              98.00   0.980394  0.9800  0.979974
1      2   0.039261              98.50   0.985000  0.9850  0.985000
2      3   0.037624              98.50   0.985059  0.9850  0.985005
3      4   0.026564              99.00   0.990183  0.9900  0.989992
4      5   0.023884              99.25   0.992604  0.9925  0.992496
5      6   0.023896              99.25   0.992515  0.9925  0.992501
6      7   0.082909              96.50   0.965348  0.9650  0.964955
7      8   0.031543              98.75   0.987517  0.9875  0.987502
8      9   0.025435              99.50   0.995046  0.9950  0.994998
9     10   0.021603              99.25   0.992604  0.9925  0.992496
```