

TestTrain

May 8, 2025

```
[3]: import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from torch.utils.data import Dataset, DataLoader
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
import os
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import json
import matplotlib.pyplot as plt
import gc
from tqdm import tqdm

# Constants
NUM_POSE_LANDMARKS = 19
NUM_HAND_LANDMARKS = 21
NUM_NODES = NUM_POSE_LANDMARKS + NUM_HAND_LANDMARKS*2 # Total nodes in the
↳graph
FEATURE_DIM = 3 # x, y coordinates + visibility

class GCNLayer(nn.Module):
    def __init__(self, in_features, out_features, dropout=0.5):
        super(GCNLayer, self).__init__()
        # self.linear = nn.Linear(in_features, out_features, bias=True)
        self.conv = nn.Conv1d(in_features, out_features, kernel_size=1,
↳bias=True)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, adj):
        # x: (batch_size * seq_len, num_nodes, in_features)
        # adj: (num_nodes, num_nodes) - shared across all samples

        # Ensure adj is on the same device as x
        if adj.device != x.device:
            adj = adj.to(x.device)
```

```

        # Expand adj to match batch dimension
        batch_size_seq = x.size(0)
        adj_expanded = adj.unsqueeze(0).expand(batch_size_seq, -1, -1)

        # Graph convolution: first aggregate neighborhood features
        x = torch.bmm(adj_expanded, x) # (batch_size * seq_len, num_nodes,
        ↪ in_features)

        # For Conv1d: input needs to be (batch, channels, length)
        # So we permute from (batch, nodes, features) to (batch, features,
        ↪ nodes)
        x = x.permute(0, 2, 1) # -> (batch_size * seq_len, in_features,
        ↪ num_nodes)

        # Apply convolution
        x = self.conv(x) # (batch_size * seq_len, out_features, num_nodes)

        # Permute back to original format
        x = x.permute(0, 2, 1) # -> (batch_size * seq_len, num_nodes,
        ↪ out_features)

        x = F.gelu(x)
        x = self.dropout(x)
        return x

class GCNBiLSTM(nn.Module):
    def __init__(self, num_nodes=NUM_NODES, in_features=FEATURE_DIM,
                  gc_hidden=64, lstm_hidden=128, num_classes=10,
                  num_gcn_layers=2, dropout=0.5, label_map=None):
        super(GCNBiLSTM, self).__init__()

        # Create multiple GCN layers
        self.gcn_layers = nn.ModuleList()
        self.gcn_layers.append(GCNLayer(in_features, gc_hidden, dropout))

        for _ in range(num_gcn_layers - 1):
            self.gcn_layers.append(GCNLayer(gc_hidden, gc_hidden, dropout))

        # Bidirectional LSTM layer
        self.lstm = nn.LSTM(
            input_size=num_nodes * gc_hidden,
            hidden_size=lstm_hidden,
            num_layers=2,
            batch_first=True,
            bidirectional=True,
            dropout=dropout if num_gcn_layers > 1 else 0

```

```

)

# Attention mechanism
self.attention = nn.Sequential(
    nn.Linear(lstm_hidden * 2, 64),
    nn.Tanh(),
    nn.Linear(64, 1)
)

# Output classification layers
self.classifier = nn.Sequential(
    nn.Linear(lstm_hidden * 2, lstm_hidden),
    nn.ReLU(),
    nn.Dropout(dropout),
    nn.Linear(lstm_hidden, num_classes)
)

self.dropout = nn.Dropout(dropout)
self.label_map = label_map
self.num_nodes = num_nodes
self.gcn_hidden = gcn_hidden

def forward(self, x, adj):
    # x shape: (batch_size, seq_len, num_nodes * in_features)
    # Reshape to (batch_size, seq_len, num_nodes, in_features)
    batch_size, seq_len, _ = x.size()
    x = x.view(batch_size, seq_len, self.num_nodes, -1)

    # Process each time step through GCN
    gcn_outputs = []
    for t in range(seq_len):
        # Get current time step data
        curr_x = x[:, t, :, :] # (batch_size, num_nodes, in_features)

        # Process through GCN layers
        for gcn_layer in self.gcn_layers:
            curr_x = gcn_layer(curr_x, adj)
            curr_x = self.dropout(curr_x)

        # Flatten node features
        # curr_x = curr_x.view(batch_size, -1) # (batch_size, num_nodes *
↪ gcn_hidden)
        curr_x = curr_x.contiguous().view(batch_size, -1)
        gcn_outputs.append(curr_x)

    # Stack outputs to (batch_size, seq_len, num_nodes * gcn_hidden)
    gcn_out = torch.stack(gcn_outputs, dim=1)

```

```

        # Process through BiLSTM
        lstm_out, _ = self.lstm(gcn_out) # (batch_size, seq_len, lstm_hidden * 2)

        # Apply attention mechanism
        attn_weights = self.attention(lstm_out).squeeze(-1) # (batch_size, seq_len)
        attn_weights = F.softmax(attn_weights, dim=1).unsqueeze(-1) # (batch_size, seq_len, 1)

        # Weighted sum of LSTM outputs
        context = torch.sum(lstm_out * attn_weights, dim=1) # (batch_size, lstm_hidden * 2)

        # Final classification
        output = self.classifier(context)

        return output

    def predict_label(self, x, adj):
        self.eval()
        with torch.no_grad():
            logits = self.forward(x, adj) # Forward pass
            pred_classes = torch.argmax(logits, dim=1) # Get the predicted class (index)

            if self.label_map is not None:
                pred_labels = [self.label_map[int(idx)] for idx in pred_classes.cpu().numpy()]
                return pred_labels
            else:
                return pred_classes

class GraphSequenceDataset(Dataset):
    def __init__(self, sequences, labels):
        self.sequences = torch.tensor(sequences, dtype=torch.float32)
        self.labels = torch.tensor(labels, dtype=torch.long)

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

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

```

```

def parse_frame(frame,):
    keypoints = []
    for part in ['pose', 'left_hand', 'right_hand']:
        for landmark in frame.get(part, []):
            keypoints.extend([landmark['x'], landmark['y'],
↪landmark['visibility']])
    return keypoints

def load_and_preprocess_data(data_dir, sequence_length=15):
    """
    Load and preprocess the JSON files into sequences.

    Args:
        data_dir: Directory containing the data
        sequence_length: Number of frames in each sequence

    Returns:
        sequences: array of shape (num_sequences, sequence_length, num_nodes *
↪features)
        sequence_labels: array of class labels
        label_encoder: fitted LabelEncoder
    """
    frame_data = []
    raw_labels = []

    # Step 1: Collect all labels
    for root, dirs, files in os.walk(data_dir):
        for file in files:
            if file.endswith(".json"):
                label = os.path.basename(os.path.dirname(os.path.join(root,
↪file)))
                raw_labels.append(label)

    # Step 2: Fit label encoder
    encoder = LabelEncoder()
    encoder.fit(raw_labels)
    label_map = {label: int(encoder.transform([label])[0]) for label in
↪set(raw_labels)}

    sequences = []
    sequence_labels = []

    # Step 3: Parse frames and assign encoded labels
    for root, dirs, files in os.walk(data_dir):

```

```

        for file in files:
            if file.endswith(".json"):
                label = os.path.basename(os.path.dirname(os.path.join(root,
↪file)))

                encoded_label = label_map[label]
                with open(os.path.join(root, file), 'r') as f:
                    frames = json.load(f)
                    features=[]
                    for frame in frames:
                        features.append(parse_frame(frame))
                        # frame_data.append([np.array(features), encoded_label])
                    sequences.append(np.stack(features))
                    sequence_labels.append(encoded_label)
idx_to_label = {v: k for k, v in label_map.items()}
label_map = idx_to_label
del idx_to_label
gc.collect()

return np.array(sequences), np.array(sequence_labels), label_map

def create_adjacency_matrix():
    """Create the adjacency matrix for the graph."""

    pose_connections = [
        # Mouth
        (9,10),
        # Left Eyes
        (1,2),(2,3),(3,7),
        # Right Eyes
        (4,5),(5,6),(6,8),
        # Nose
        (0,4),(0,1),
        # Shoulders
        (11, 12),
        # Connect shoulders to hip
        (11, 17), (12, 18),
        # Connect hip points
        (17, 18),
        # Left arm
        (11, 13), (13, 15),
        # Right arm
        (12, 14), (14, 16)
    ]

    hand_connections = [
        # Thumb

```

```

(0, 1), (1, 2), (2, 3), (3, 4),
# Index finger
(0, 5), (5, 6), (6, 7), (7, 8),
# Middle finger
(0, 9), (9, 10), (10, 11), (11, 12),
# Ring finger
(0, 13), (13, 14), (14, 15), (15, 16),
# Pinky
(0, 17), (17, 18), (18, 19), (19, 20),
# Palm connections
(5, 9), (9, 13), (13, 17)
]

def create_adj_matrix(num_nodes, connections):
    adj_matrix = np.zeros((num_nodes, num_nodes))
    for i, j in connections:
        adj_matrix[i, j] = 1
        adj_matrix[j, i] = 1
    # Add self-loops
    for i in range(num_nodes):
        adj_matrix[i, i] = 1
    return adj_matrix

pose_adj_matrix = create_adj_matrix(NUM_POSE_LANDMARKS, pose_connections)
left_hand_adj_matrix = create_adj_matrix(NUM_HAND_LANDMARKS,
↪hand_connections)
right_hand_adj_matrix = create_adj_matrix(NUM_HAND_LANDMARKS,
↪hand_connections)

# Calculate the total number of nodes
total_nodes = NUM_POSE_LANDMARKS + NUM_HAND_LANDMARKS + NUM_HAND_LANDMARKS

# Initialize a global adjacency matrix
global_adj_matrix = np.zeros((total_nodes, total_nodes))

# start_pose = NUM_FACE_LANDMARKS
start_pose=0
end_pose = start_pose + NUM_POSE_LANDMARKS
global_adj_matrix[start_pose:end_pose, start_pose:end_pose] =
↪pose_adj_matrix

start_lh = end_pose
end_lh = start_lh + NUM_HAND_LANDMARKS
global_adj_matrix[start_lh:end_lh, start_lh:end_lh] = left_hand_adj_matrix

start_rh = end_lh
end_rh = start_rh + NUM_HAND_LANDMARKS

```

```

global_adj_matrix[start_rh:end_rh, start_rh:end_rh] = right_hand_adj_matrix

# Connect pose to hands
pose_hand_connections = [
    (start_pose + 15, start_lh), # Left hand wrist to left hand base
    (start_pose + 16, start_rh), # Right hand wrist to right hand base
]
for i, j in pose_hand_connections:
    global_adj_matrix[i, j] = 1
    global_adj_matrix[j, i] = 1

# Normalize adjacency matrix ( $D^{-0.5} * A * D^{-0.5}$ )
# Add identity matrix to include self-connections
adj_matrix = global_adj_matrix + np.eye(total_nodes)

# Calculate degree matrix
degree_matrix = np.diag(np.sum(adj_matrix, axis=1))

#  $D^{-0.5}$ 
deg_inv_sqrt = np.linalg.inv(np.sqrt(degree_matrix))

# Normalized adjacency matrix
normalized_adj_matrix = deg_inv_sqrt @ adj_matrix @ deg_inv_sqrt

return torch.FloatTensor(normalized_adj_matrix)

def train_model(model, train_loader, val_loader, adj_matrix, num_epochs=50,
    lr=0.001,
    weight_decay=1e-5, patience=10, model_save_path='best_model.pt'):
    """
    Train the GCNBiLSTM model

    Args:
        model: GCNBiLSTM model
        train_loader: DataLoader for training data
        val_loader: DataLoader for validation data
        adj_matrix: Normalized adjacency matrix
        num_epochs: Number of training epochs
        lr: Learning rate
        weight_decay: Weight decay factor
        patience: Early stopping patience
        model_save_path: Path to save best model

    Returns:
        model: Trained model
        train_losses: List of training losses
    """

```



```

    val_losses: List of validation losses
    train_accs: List of training accuracies
    val_accs: List of validation accuracies
    """
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(f"Using device: {device}")

    model = model.to(device)
    adj_matrix = adj_matrix.to(device)

    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr,
    ↪weight_decay=weight_decay)

    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min',
    ↪patience=5, factor=0.5)

    train_losses = []
    val_losses = []
    train_accs = []
    val_accs = []

    best_val_loss = float('inf')
    early_stop_counter = 0

    for epoch in range(num_epochs):
        # Training phase
        model.train()
        train_loss = 0.0
        correct = 0
        total = 0

        progress_bar = tqdm(train_loader, desc=f"Epoch {epoch+1}/{num_epochs}",
    ↪[Train])
        for batch_idx, (inputs, targets) in enumerate(progress_bar):
            inputs, targets = inputs.to(device), targets.to(device)

            optimizer.zero_grad()
            outputs = model(inputs, adj_matrix)

            loss = criterion(outputs, targets)
            loss.backward()

            # Optional: gradient clipping
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=5.0)

```

```

optimizer.step()

train_loss += loss.item()
_, predicted = outputs.max(1)
total += targets.size(0)
correct += predicted.eq(targets).sum().item()

progress_bar.set_postfix({
    'loss': train_loss/(batch_idx+1),
    'acc': 100.*correct/total
})

train_loss = train_loss / len(train_loader)
train_acc = 100. * correct / total
train_losses.append(train_loss)
train_accs.append(train_acc)

# Validation phase
model.eval()
val_loss = 0.0
correct = 0
total = 0

with torch.no_grad():
    for batch_idx, (inputs, targets) in enumerate(val_loader):
        inputs, targets = inputs.to(device), targets.to(device)
        outputs = model(inputs, adj_matrix)

        loss = criterion(outputs, targets)
        val_loss += loss.item()

        _, predicted = outputs.max(1)
        total += targets.size(0)
        correct += predicted.eq(targets).sum().item()

val_loss = val_loss / len(val_loader)
val_acc = 100. * correct / total
val_losses.append(val_loss)
val_accs.append(val_acc)

# Update learning rate
scheduler.step(val_loss)

print(f"Epoch {epoch+1}/{num_epochs} - "
      f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.2f}% - "
      f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.2f}%")

```

```

        # Save best model
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            # Save only the model state dict
            torch.save({'model_state_dict':model.state_dict(), 'label_map':model.
↪label_map}, model_save_path)

            early_stop_counter = 0
            print(f"Saved best model to {model_save_path}")
        else:
            early_stop_counter += 1

        # Early stopping
        if early_stop_counter >= patience:
            print(f"Early stopping triggered after {epoch+1} epochs")
            break

    # Load best model
    checkpoint = torch.load(model_save_path)
    model.load_state_dict(checkpoint['model_state_dict'])
    model.label_map = checkpoint['label_map']
    return model, train_losses, val_losses, train_accs, val_accs

def evaluate_model(model, test_loader, adj_matrix):
    """
    Evaluate the model on test data

    Args:
        model: Trained GCNBiLSTM model
        test_loader: DataLoader for test data
        adj_matrix: Normalized adjacency matrix

    Returns:
        test_acc: Test accuracy
        predictions: Predicted labels
        true_labels: True labels
    """
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = model.to(device)
    adj_matrix = adj_matrix.to(device)

    model.eval()
    correct = 0
    total = 0

```

```

all_preds = []
all_targets = []

with torch.no_grad():
    for inputs, targets in test_loader:
        inputs, targets = inputs.to(device), targets.to(device)
        outputs = model(inputs, adj_matrix)

        _, predicted = outputs.max(1)
        total += targets.size(0)
        correct += predicted.eq(targets).sum().item()

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

test_acc = 100. * correct / total
print(f"Test Accuracy: {test_acc:.2f}%")

return test_acc, np.array(all_preds), np.array(all_targets)

def plot_results(train_losses, val_losses, train_accs, val_accs):
    """
    Plot training and validation metrics

    Args:
        train_losses: List of training losses
        val_losses: List of validation losses
        train_accs: List of training accuracies
        val_accs: List of validation accuracies
    """
    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label='Train Loss')
    plt.plot(val_losses, label='Val Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.title('Training and Validation Loss')

    plt.subplot(1, 2, 2)
    plt.plot(train_accs, label='Train Accuracy')
    plt.plot(val_accs, label='Val Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy (%)')
    plt.legend()

```

```

plt.title('Training and Validation Accuracy')

plt.tight_layout()
plt.savefig('training_history.png')
plt.show()

def main():
    torch.serialization.add_safe_globals([LabelEncoder])
    # Set random seeds for reproducibility
    SEED = 42
    torch.manual_seed(SEED)
    np.random.seed(SEED)

    # 1. Load and preprocess data
    data_dir = "data" # Update with your data directory
    test_dir = "test_data"
    sequences, sequence_labels, label_map = load_and_preprocess_data(data_dir)

    print(f"Loaded {len(sequences)} sequences with shape {sequences.shape}")
    print(f'Sequence Label {len(sequence_labels)}')
    print(f"Number of classes: {len(label_map)}")

    # 2. Create adjacency matrix

    adj_matrix = create_adjacency_matrix()
    print(f'Unique Label : {len(np.unique(sequence_labels))}')
    X_train, X_val, y_train, y_val = train_test_split(
        sequences, sequence_labels, test_size=0.4, random_state=SEED,
↪ stratify=sequence_labels
    )
    print(f'Unique Train : {len(np.unique(y_train))}')
    print(f'Unique Val : {len(np.unique(y_val))}')
    X_val, X_test, y_val, y_test = train_test_split(
        X_val, y_val, test_size=0.5, random_state=SEED, stratify=y_val
    )
    print(f'Unique Train : {len(np.unique(y_train))}')
    print(f'Unique Test : {len(np.unique(y_test))}')
    # X_test, y_test, _ = load_and_preprocess_data(test_dir)

    # 4. Create datasets and dataloaders
    train_dataset = GraphSequenceDataset(X_train, y_train)
    val_dataset = GraphSequenceDataset(X_val, y_val)
    test_dataset = GraphSequenceDataset(X_test, y_test)

    train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)

```

```

val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

# 5. Create and train the model
num_classes = len(label_map)
model = GCNBiLSTM(
    num_nodes=NUM_NODES,
    in_features=FEATURE_DIM,
    gcn_hidden=256,
    lstm_hidden=512,
    num_classes=num_classes,
    num_gcn_layers=2,
    dropout=0.3,
    label_map=label_map
)

# 6. Train the model
trained_model, train_losses, val_losses, train_accs, val_accs = train_model(
    model, train_loader, val_loader, adj_matrix,
    num_epochs=100, lr=0.001, weight_decay=5e-4,
    patience=15, model_save_path='best_gcn_bilstm_model.pt'
)

# 7. Evaluate the model
test_acc, predictions, true_labels = evaluate_model(trained_model,
↳test_loader, adj_matrix)

# 8. Plot results
plot_results(train_losses, val_losses, train_accs, val_accs)

# 9. Print classification report

print("\nClassification Report:")
actual_classes = np.unique(true_labels)
print(actual_classes)
class_names = [label_map[int(idx)] for idx in actual_classes]
print(classification_report(true_labels, predictions,
↳target_names=class_names))

# 10. Plot confusion matrix
plt.figure(figsize=(12, 10))
cm = confusion_matrix(true_labels, predictions)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names,
↳yticklabels=class_names)
plt.xlabel('Predicted')
plt.ylabel('True')

```

```
plt.title('Confusion Matrix')
plt.tight_layout()
plt.savefig('confusion_matrix.png')
plt.show()
```

```
[4]: main()
```

```
Loaded 1691 sequences with shape (1691, 3, 183)
Sequence Label 1691
Number of classes: 26
Unique Label : 26
Unique Train : 26
Unique Val : 26
Unique Train : 26
Unique Test : 26
Using device: cuda

Epoch 1/100 [Train]: 100%|      | 32/32 [00:11<00:00, 2.73it/s, loss=2.97,
acc=7]

Epoch 1/100 - Train Loss: 2.9659, Train Acc: 7.00% - Val Loss: 2.7701, Val Acc:
9.47%
Saved best model to best_gcn_bilstm_model.pt

Epoch 2/100 [Train]: 100%|      | 32/32 [00:01<00:00, 17.27it/s, loss=2.75,
acc=10.2]

Epoch 2/100 - Train Loss: 2.7524, Train Acc: 10.16% - Val Loss: 2.7486, Val Acc:
12.43%
Saved best model to best_gcn_bilstm_model.pt

Epoch 3/100 [Train]: 100%|      | 32/32 [00:01<00:00, 16.66it/s, loss=2.74,
acc=9.17]

Epoch 3/100 - Train Loss: 2.7415, Train Acc: 9.17% - Val Loss: 2.7350, Val Acc:
10.36%
Saved best model to best_gcn_bilstm_model.pt

Epoch 4/100 [Train]: 100%|      | 32/32 [00:01<00:00, 18.59it/s, loss=2.73,
acc=10.7]

Epoch 4/100 - Train Loss: 2.7292, Train Acc: 10.65% - Val Loss: 2.7264, Val Acc:
10.36%
Saved best model to best_gcn_bilstm_model.pt

Epoch 5/100 [Train]: 100%|      | 32/32 [00:01<00:00, 19.64it/s, loss=2.73,
acc=7.79]

Epoch 5/100 - Train Loss: 2.7330, Train Acc: 7.79% - Val Loss: 2.7344, Val Acc:
10.95%

Epoch 6/100 [Train]: 100%|      | 32/32 [00:01<00:00, 17.99it/s, loss=2.72,
acc=9.17]
```

Epoch 6/100 - Train Loss: 2.7176, Train Acc: 9.17% - Val Loss: 2.7154, Val Acc: 8.88%
Saved best model to best_gcn_bilstm_model.pt

Epoch 7/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.08it/s, loss=2.72, acc=9.37]

Epoch 7/100 - Train Loss: 2.7154, Train Acc: 9.37% - Val Loss: 2.7190, Val Acc: 10.95%

Epoch 8/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.41it/s, loss=2.71, acc=10.7]

Epoch 8/100 - Train Loss: 2.7106, Train Acc: 10.65% - Val Loss: 2.7091, Val Acc: 10.36%
Saved best model to best_gcn_bilstm_model.pt

Epoch 9/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.62it/s, loss=2.69, acc=12.4]

Epoch 9/100 - Train Loss: 2.6872, Train Acc: 12.43% - Val Loss: 2.6930, Val Acc: 8.58%
Saved best model to best_gcn_bilstm_model.pt

Epoch 10/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.18it/s, loss=2.54, acc=16.1]

Epoch 10/100 - Train Loss: 2.5385, Train Acc: 16.07% - Val Loss: 2.3782, Val Acc: 23.67%
Saved best model to best_gcn_bilstm_model.pt

Epoch 11/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.10it/s, loss=2.37, acc=22.9]

Epoch 11/100 - Train Loss: 2.3687, Train Acc: 22.88% - Val Loss: 2.2107, Val Acc: 23.96%
Saved best model to best_gcn_bilstm_model.pt

Epoch 12/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.55it/s, loss=2.18, acc=26.6]

Epoch 12/100 - Train Loss: 2.1766, Train Acc: 26.63% - Val Loss: 1.9973, Val Acc: 33.14%
Saved best model to best_gcn_bilstm_model.pt

Epoch 13/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.50it/s, loss=1.98, acc=35]

Epoch 13/100 - Train Loss: 1.9771, Train Acc: 35.01% - Val Loss: 1.8794, Val Acc: 39.05%
Saved best model to best_gcn_bilstm_model.pt

Epoch 14/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.58it/s, loss=1.82, acc=38.4]

Epoch 14/100 - Train Loss: 1.8156, Train Acc: 38.36% - Val Loss: 1.8240, Val Acc: 40.83%
Saved best model to best_gcn_bilstm_model.pt

Epoch 15/100 [Train]: 100%| | 32/32 [00:01<00:00, 20.01it/s, loss=1.75, acc=40.7]

Epoch 15/100 - Train Loss: 1.7502, Train Acc: 40.73% - Val Loss: 1.6359, Val Acc: 45.56%
Saved best model to best_gcn_bilstm_model.pt

Epoch 16/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.55it/s, loss=1.59, acc=47]

Epoch 16/100 - Train Loss: 1.5896, Train Acc: 47.04% - Val Loss: 1.5452, Val Acc: 48.22%
Saved best model to best_gcn_bilstm_model.pt

Epoch 17/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.93it/s, loss=1.5, acc=49.2]

Epoch 17/100 - Train Loss: 1.5040, Train Acc: 49.21% - Val Loss: 1.4690, Val Acc: 52.07%
Saved best model to best_gcn_bilstm_model.pt

Epoch 18/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.96it/s, loss=1.42, acc=50.2]

Epoch 18/100 - Train Loss: 1.4212, Train Acc: 50.20% - Val Loss: 1.4201, Val Acc: 53.55%
Saved best model to best_gcn_bilstm_model.pt

Epoch 19/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.32it/s, loss=1.36, acc=54.7]

Epoch 19/100 - Train Loss: 1.3588, Train Acc: 54.73% - Val Loss: 1.3615, Val Acc: 55.62%
Saved best model to best_gcn_bilstm_model.pt

Epoch 20/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.94it/s, loss=1.31, acc=54.5]

Epoch 20/100 - Train Loss: 1.3114, Train Acc: 54.54% - Val Loss: 1.3324, Val Acc: 57.10%
Saved best model to best_gcn_bilstm_model.pt

Epoch 21/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.51it/s, loss=1.28, acc=58.6]

Epoch 21/100 - Train Loss: 1.2807, Train Acc: 58.58% - Val Loss: 1.3232, Val Acc: 56.21%
Saved best model to best_gcn_bilstm_model.pt

Epoch 22/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.72it/s, loss=1.17, acc=60.7]

Epoch 22/100 - Train Loss: 1.1713, Train Acc: 60.75% - Val Loss: 1.3261, Val Acc: 60.65%

Epoch 23/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.99it/s, loss=1.12, acc=62.2]

Epoch 23/100 - Train Loss: 1.1240, Train Acc: 62.23% - Val Loss: 1.1796, Val Acc: 64.20%

Saved best model to best_gcn_bilstm_model.pt

Epoch 24/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.74it/s, loss=1.05, acc=63.4]

Epoch 24/100 - Train Loss: 1.0502, Train Acc: 63.41% - Val Loss: 1.3624, Val Acc: 64.20%

Epoch 25/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.31it/s, loss=1.1, acc=62.6]

Epoch 25/100 - Train Loss: 1.0998, Train Acc: 62.62% - Val Loss: 1.1467, Val Acc: 64.20%

Saved best model to best_gcn_bilstm_model.pt

Epoch 26/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.25it/s, loss=0.955, acc=66.6]

Epoch 26/100 - Train Loss: 0.9552, Train Acc: 66.57% - Val Loss: 1.1718, Val Acc: 64.20%

Epoch 27/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.35it/s, loss=0.909, acc=69.5]

Epoch 27/100 - Train Loss: 0.9093, Train Acc: 69.53% - Val Loss: 1.1474, Val Acc: 67.75%

Epoch 28/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.53it/s, loss=0.901, acc=69.1]

Epoch 28/100 - Train Loss: 0.9013, Train Acc: 69.13% - Val Loss: 1.0788, Val Acc: 65.68%

Saved best model to best_gcn_bilstm_model.pt

Epoch 29/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.62it/s, loss=0.88, acc=69.4]

Epoch 29/100 - Train Loss: 0.8797, Train Acc: 69.43% - Val Loss: 1.1237, Val Acc: 68.34%

Epoch 30/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.28it/s, loss=0.844, acc=70.2]

Epoch 30/100 - Train Loss: 0.8442, Train Acc: 70.22% - Val Loss: 0.9920, Val Acc: 67.75%

Saved best model to best_gcn_bilstm_model.pt

Epoch 31/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.26it/s,
loss=0.849, acc=69.7]

Epoch 31/100 - Train Loss: 0.8490, Train Acc: 69.72% - Val Loss: 1.0086, Val
Acc: 68.93%

Epoch 32/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.02it/s,
loss=0.796, acc=72.4]

Epoch 32/100 - Train Loss: 0.7957, Train Acc: 72.39% - Val Loss: 0.9738, Val
Acc: 69.82%

Saved best model to best_gcn_bilstm_model.pt

Epoch 33/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.74it/s,
loss=0.759, acc=75.2]

Epoch 33/100 - Train Loss: 0.7592, Train Acc: 75.25% - Val Loss: 1.0250, Val
Acc: 71.60%

Epoch 34/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.65it/s,
loss=0.786, acc=75.3]

Epoch 34/100 - Train Loss: 0.7857, Train Acc: 75.35% - Val Loss: 1.0285, Val
Acc: 70.41%

Epoch 35/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.48it/s,
loss=0.721, acc=76.1]

Epoch 35/100 - Train Loss: 0.7207, Train Acc: 76.13% - Val Loss: 1.0782, Val
Acc: 71.01%

Epoch 36/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.43it/s,
loss=0.715, acc=75.3]

Epoch 36/100 - Train Loss: 0.7148, Train Acc: 75.35% - Val Loss: 1.0344, Val
Acc: 67.75%

Epoch 37/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.32it/s,
loss=0.697, acc=76.1]

Epoch 37/100 - Train Loss: 0.6969, Train Acc: 76.13% - Val Loss: 1.0267, Val
Acc: 70.12%

Epoch 38/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.34it/s, loss=0.67,
acc=78.2]

Epoch 38/100 - Train Loss: 0.6697, Train Acc: 78.21% - Val Loss: 1.0334, Val
Acc: 70.12%

Epoch 39/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.43it/s,
loss=0.589, acc=79.2]

Epoch 39/100 - Train Loss: 0.5888, Train Acc: 79.19% - Val Loss: 0.8990, Val
Acc: 73.67%

Saved best model to best_gcn_bilstm_model.pt

Epoch 40/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.02it/s,
loss=0.511, acc=82.9]

Epoch 40/100 - Train Loss: 0.5112, Train Acc: 82.94% - Val Loss: 0.9481, Val
Acc: 69.82%

Epoch 41/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.65it/s,
loss=0.484, acc=81.9]

Epoch 41/100 - Train Loss: 0.4841, Train Acc: 81.85% - Val Loss: 0.9214, Val
Acc: 71.30%

Epoch 42/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.92it/s,
loss=0.451, acc=83.4]

Epoch 42/100 - Train Loss: 0.4506, Train Acc: 83.43% - Val Loss: 0.9144, Val
Acc: 72.19%

Epoch 43/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.03it/s,
loss=0.483, acc=83]

Epoch 43/100 - Train Loss: 0.4828, Train Acc: 83.04% - Val Loss: 0.9912, Val
Acc: 71.30%

Epoch 44/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.12it/s,
loss=0.437, acc=84.2]

Epoch 44/100 - Train Loss: 0.4368, Train Acc: 84.22% - Val Loss: 0.9151, Val
Acc: 71.60%

Epoch 45/100 [Train]: 100%| | 32/32 [00:01<00:00, 20.10it/s,
loss=0.435, acc=84.1]

Epoch 45/100 - Train Loss: 0.4352, Train Acc: 84.12% - Val Loss: 0.9621, Val
Acc: 71.89%

Epoch 46/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.78it/s,
loss=0.396, acc=86.5]

Epoch 46/100 - Train Loss: 0.3958, Train Acc: 86.49% - Val Loss: 0.9532, Val
Acc: 73.37%

Epoch 47/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.35it/s,
loss=0.366, acc=86.9]

Epoch 47/100 - Train Loss: 0.3662, Train Acc: 86.88% - Val Loss: 0.9564, Val
Acc: 72.49%

Epoch 48/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.71it/s, loss=0.36,
acc=88.4]

Epoch 48/100 - Train Loss: 0.3598, Train Acc: 88.36% - Val Loss: 0.9746, Val
Acc: 71.30%

Epoch 49/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.65it/s,
loss=0.333, acc=87.9]

Epoch 49/100 - Train Loss: 0.3330, Train Acc: 87.87% - Val Loss: 0.9566, Val Acc: 72.49%

Epoch 50/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.75it/s, loss=0.31, acc=89.4]

Epoch 50/100 - Train Loss: 0.3099, Train Acc: 89.45% - Val Loss: 0.9731, Val Acc: 72.19%

Epoch 51/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.89it/s, loss=0.325, acc=88.7]

Epoch 51/100 - Train Loss: 0.3247, Train Acc: 88.66% - Val Loss: 0.9886, Val Acc: 71.01%

Epoch 52/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.63it/s, loss=0.305, acc=89.2]

Epoch 52/100 - Train Loss: 0.3047, Train Acc: 89.15% - Val Loss: 0.9847, Val Acc: 72.19%

Epoch 53/100 [Train]: 100%| | 32/32 [00:01<00:00, 18.89it/s, loss=0.286, acc=90.4]

Epoch 53/100 - Train Loss: 0.2860, Train Acc: 90.43% - Val Loss: 0.9626, Val Acc: 73.67%

Epoch 54/100 [Train]: 100%| | 32/32 [00:01<00:00, 19.03it/s, loss=0.277, acc=89.7]

Epoch 54/100 - Train Loss: 0.2770, Train Acc: 89.74% - Val Loss: 1.0024, Val Acc: 71.89%

Early stopping triggered after 54 epochs

Test Accuracy: 73.16%



Classification Report:

[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25]

	precision	recall	f1-score	support
A	1.00	0.62	0.77	8
B	0.50	0.50	0.50	10
C	0.52	0.78	0.62	18
D	0.45	0.56	0.50	9
E	0.76	0.89	0.82	18
F	0.00	0.00	0.00	6
G	0.86	0.67	0.75	9
H	0.62	0.56	0.59	9
I	0.95	0.95	0.95	22
J	0.75	0.86	0.80	21
K	1.00	0.82	0.90	11
L	0.57	0.68	0.62	19
M	1.00	0.43	0.60	7
N	0.33	0.33	0.33	6
O	0.87	0.91	0.89	22
P	0.50	0.11	0.18	9
Q	0.78	0.78	0.78	9
R	0.46	0.32	0.38	19
S	0.80	0.73	0.76	11
T	0.61	0.85	0.71	13
U	0.82	1.00	0.90	18
V	0.88	0.88	0.88	16
W	0.83	0.88	0.86	17
X	0.57	0.50	0.53	8
Y	0.60	0.60	0.60	5
Z	1.00	1.00	1.00	19
accuracy			0.73	339
macro avg	0.69	0.66	0.66	339
weighted avg	0.73	0.73	0.72	339

