GNN + Transformer Model: Code Documentation

This document provides detailed explanations of the code implementation for the GNN + Transformer model for keypoint-based score generation.

Table of Contents

Data Processing Pipeline

MultiPickleProcessor Class

The `MultiPickleProcessor` class (in `multi_pickle_processor.py`) is responsible for:

1. Loading data from multiple pickle directories with different formats
2. Unifying the data into a consistent format
3. Generating a segment database that combines body, hand, and object data

Key methods:

python

```python
def load_pickle_files(self):
    # Loads pickle files from each directory and handles different data formats
    self._load_body_pickle_files()  # Load body keypoint data
    self._load_openpose_pickle_files()  # Load alternative OpenPose format
    self._load_hand_pickle_files()  # Load hand keypoint data
    self._load_object_pickle_files()  # Load object location data

def build_segment_database(self, therapist_labels_path=None):
    # Combines data from different sources and creates segments
    # Each segment contains body keypoints, hand keypoints, object locations, and label

def save_segment_database(self, filename='segment_database.pkl'):
    # Saves the combined segment database to disk
```

The processor handles different data formats by providing specialized methods for each data source. It adapts to the specific format of each pickle file and unifies the data into a consistent segment database.

Data Loading and Graph Construction

KeypointDataset Class

The `KeypointDataset` class (in `data_loader.py`) is responsible for:

1. Loading segments from the database
2. Converting keypoint data into graph representations
3. Managing different data modalities (body, hand, object)

Graph construction is a critical part of the process. For each frame:

python

```python
def _create_frame_graph(self, body_keypoints, hand_keypoints, object_locations):
    # Body keypoint indices for graph edges (OpenPose format)
    body_edges = [
        # Torso connections
        (0, 1), (1, 2), (2, 3), (3, 4),  # Neck to right arm
        (1, 5), (5, 6), (6, 7),          # Neck to left arm
        # ... other connections
    ]

    # Hand keypoint indices for graph edges (MediaPipe format)
    hand_edges = [
        # Thumb connections
        (0, 1), (1, 2), (2, 3), (3, 4),
        # ... other finger connections
    ]

    # Create edges between nodes
    edge_index_list = []

    # Add body keypoint edges
    # ... code to add body edges

    # Add hand keypoint edges
    # ... code to add hand edges

    # Connect hand to body
    # ... code to connect hand and body nodes

    # Connect objects to hands and body
    # ... code to connect object nodes to relevant body and hand nodes

    # Create PyTorch Geometric Data object
    graph = Data(x=x, edge_index=edge_index)

    return graph
```

This method creates a graph where:

- Nodes represent keypoints (body joints, hand points, object locations)
- Edges represent connections between keypoints
- Body-hand connections link the wrist of the body to the wrist of the hand

- Object-body connections link objects to relevant body keypoints (e.g., hands)
- Object-hand connections link objects to relevant hand keypoints (e.g., fingertips)

GNN Encoder Architecture

The `GNNEncoder` class (in `gnn_transformer_updated.py`) implements a Graph Convolutional Network (GCN) to process spatial relationships:

python

```python
class GNNEncoder(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, dropout=0.2):
        # Initialize GNN layers
        self.conv1 = GCNConv(input_dim, hidden_dim)
        self.conv2 = GCNConv(hidden_dim, hidden_dim)
        self.conv3 = GCNConv(hidden_dim, output_dim)

        # Normalization and dropout
        self.norm1 = nn.LayerNorm(hidden_dim)
        self.norm2 = nn.LayerNorm(hidden_dim)
        self.dropout_layer = nn.Dropout(dropout)

    def forward(self, x, edge_index, batch):
        # First GNN layer
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = self.norm1(x)
        x = self.dropout_layer(x)

        # Second GNN layer
        x = self.conv2(x, edge_index)
        x = F.relu(x)
        x = self.norm2(x)
        x = self.dropout_layer(x)

        # Third GNN layer
        x = self.conv3(x, edge_index)

        # Global pooling to get graph-level embeddings
        x = torch.cat([
            torch.mean(x[batch == i], dim=0, keepdim=True)
            for i in range(batch.max().item() + 1)
        ], dim=0)

        return x
```

The GNN encoder:

1. Takes node features, edge indices, and batch assignments as input
2. Applies multiple GCN layers with normalization and dropout
3. Uses mean pooling to aggregate node-level features into graph-level embeddings

4. Returns a fixed-size embedding for each graph

Transformer Encoder Architecture

The `TransformerEncoder` class (in `gnn_transformer_updated.py`) processes temporal relationships across frames:

python

```
class TransformerEncoder(nn.Module):
    def __init__(self, input_dim, num_heads=4, num_layers=4, dropout=0.2):
        # Transformer encoder layer
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=input_dim,
            nhead=num_heads,
            dim_feedforward=4 * input_dim,
            dropout=dropout,
            activation='relu',
            batch_first=True
        )

        # Transformer encoder
        self.transformer = nn.TransformerEncoder(
            encoder_layer=encoder_layer,
            num_layers=num_layers
        )

        # Position encoding
        self.position_encoding = PositionalEncoding(
            d_model=input_dim,
            dropout=dropout,
            max_len=1000
        )

    def forward(self, x, mask=None):
        # Add positional encoding
        x = self.position_encoding(x)

        # Apply transformer encoder
        # ... handling attention mask if provided
        x = self.transformer(x)

        return x
```

The transformer encoder:

1. Takes a sequence of GNN embeddings as input
2. Adds positional encoding to provide temporal information
3. Applies multiple transformer encoder layers
4. Returns a sequence of encoded representations

Training and Evaluation Process

The training process (`train_model` function in `main_script.py`) includes:

1. Data loading and preparation
2. Model creation and initialization
3. Training loop with validation
4. Learning rate scheduling
5. Checkpoint saving
6. Visualization and evaluation

Key components of the training loop:

python

```python
# Training loop
for epoch in range(start_epoch, epochs):
    # Train
    train_loss, train_acc = train_epoch(model, train_loader, criterion, optimizer, device)

    # Validate
    val_loss, val_acc, val_f1, _, _ = validate(model, val_loader, criterion, device)

    # Update learning rate
    scheduler.step(val_acc)

    # Save best model
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        best_epoch = epoch
        torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'val_acc': val_acc,
            'val_f1': val_f1
        }, os.path.join(output_dir, 'gnn_transformer_best.pt'))
```

The evaluation process (`test` function) calculates:

- Accuracy
- F1 score
- Confusion matrix
- Detailed classification report

Visualization Utilities

The model includes visualizations for:

Keypoint Visualization

python

```python
def visualize_keypoints(data_loader, output_dir, num_samples=5):
    # For each sample, show keypoints at different frames
    for i, sample in enumerate(samples):
        # Get graph sequence and create figure
        # Plot nodes and edges for each frame
        # Save visualization
```

This function visualizes keypoints for selected frames (first, middle, and last) showing:

- Nodes as points
- Edges as lines connecting keypoints

Attention Weight Visualization

python

```python
def visualize_attention(model, data_loader, device, output_dir, num_samples=5):
    # Get attention weights from model
    attention_weights = model.get_attention_weights(sample_graphs, sample_seq_length)

    # For each sample, visualize attention weights
    for i, sample in enumerate(samples):
        # Create figure with subplots for each layer
        # Plot attention matrix for each transformer layer
        # Save visualization
```

This function visualizes attention weights as heatmaps showing:

- Which frames attend to which other frames
- Patterns of attention across the sequence
- Different attention patterns in each transformer layer

Cross-Validation Implementation

The cross-validation process (`cross_validate` function in `main_script.py`) includes:

1. Data splitting into folds with stratification
2. Training and evaluation on each fold
3. Aggregation of metrics across folds

Key components:

python

```python
# Split each class into folds
for label in unique_labels:
    label_indices = [i for i, l in enumerate(labels) if l == label]
    np.random.shuffle(label_indices)
    fold_size = len(label_indices) // num_folds

    for fold in range(num_folds):
        start_idx = fold * fold_size
        end_idx = (fold + 1) * fold_size if fold < num_folds - 1 else len(label_indices)
        fold_indices.append((fold, label_indices[start_idx:end_idx]))

# Train and evaluate on each fold
for fold in range(num_folds):
    # Create train and test sets for this fold
    test_ids = fold_data[fold]
    train_ids = [sid for f in range(num_folds) if f != fold for sid in fold_data[f]]

    # Train on this fold
    accuracy, f1, _ = train_model(**train_kwargs)

    # Store metrics
    fold_metrics.append({
        'fold': fold + 1,
        'accuracy': accuracy,
        'f1_score': f1
    })

# Calculate average metrics
avg_accuracy = np.mean([m['accuracy'] for m in fold_metrics])
avg_f1 = np.mean([m['f1_score'] for m in fold_metrics])
```

Cross-validation ensures:

- Each example is used for both training and testing
- Results are representative of model performance
- There's no data leakage between folds

Pipeline Integration

The `run_pipeline` function (in `run_pipeline.py`) integrates all the components:

python

```python
def run_pipeline(mode='process_and_train'):
    # Configuration
    config = {
        # Data paths, model parameters, etc.
        # ...
    }

    # Process pickle files if needed
    if mode in ['process_only', 'process_and_train', 'cross_validate']:
        processor = MultiPickleProcessor(...)
        processor.process(...)

    # Train model if needed
    if mode in ['train_only', 'process_and_train']:
        train_model(...)

    # Run cross-validation if needed
    if mode == 'cross_validate':
        cross_validate(...)

    return {
        'segment_db_path': segment_db_path,
        'model_output_dir': config['model_output_dir']
    }
```

This function provides a simplified interface to run the complete pipeline or specific steps as needed.

Memory and Performance Considerations

- **Batch Processing**: The model processes data in batches to reduce memory requirements
- **Graph Batching**: PyTorch Geometric's `Batch` class efficiently handles multiple graphs
- **Sequence Length**: Limiting sequence length helps control memory usage
- **Checkpointing**: Regular saving of model checkpoints prevents loss of progress
- **Parallel Processing**: Data loading is parallelized with multiple workers

Debug Tips

If you encounter issues:

1. **Data Loading**: Check the structure of pickle files and ensure they're being loaded correctly

2. **Graph Construction**: Verify the edge connections are valid
3. **Training Instability**: Reduce learning rate or add more normalization
4. **Memory Issues**: Reduce batch size or sequence length
5. **CUDA Out of Memory**: Use smaller model dimensions or process on CPU