Midterm Task Komputasi Intelegensia

Fauzan Adzhima Alamsyah
1906376804

Abstract

This paper explores the challenges of combining YOLO object detection models with Transformers for behavior analysis, focusing on the technical difficulties of integrating YOLO's real-time detection with Transformer models for tracking and temporal sequence analysis. We discuss how YOLO's bounding box sequences can serve as input for Transformer-based temporal behavior prediction, outlining necessary adjustments to ensure compatibility and effectiveness. Additionally, we evaluate the strengths of Transformers in predicting object behaviors over time, leveraging their self-attention mechanism to analyze movement patterns, compared to traditional RNN-based models. Scenarios such as detecting suspicious activities in surveillance footage and classifying vehicle speed and direction in traffic are explored. Finally, we address the impact of model optimization techniques, such as quantization and input size reduction, on real-time performance. We examine how these optimizations balance detection accuracy and effective temporal analysis in constrained environments like mobile devices and edge computing platforms.

1 Challenges of Combining YOLO with Transformers for Behavior Analysis

1.1 Data Compatibility and Preprocessing YOLO Output for Transformers

1.1.1 YOLO Output as Temporal Sequence Data

YOLO outputs bounding box coordinates and class labels for each detected object in a frame, represented as:

$$\mathbf{B} = \{(x, y, w, h, c)\} \tag{1}$$

where x and y denote the bounding box center coordinates, w and h the width and height, and c the class label. To feed these data into a Transformer, the bounding box information needs to be structured into a temporal sequence:

$$\mathbf{S} = \{\mathbf{B}_1, \mathbf{B}_2, \ldots, \mathbf{B}_T\} \tag{2}$$

where T is the number of frames.

1.1.2 Temporal Sequence Adjustments

Transformers require sequences of fixed length, yet YOLO outputs vary in length across frames. To address this, padding or masking is applied:

$$\mathbf{S}_{\text{padded}} = \{\mathbf{B}_1, \mathbf{B}_2, \ldots, \mathbf{B}_T, 0, \ldots, 0\} \tag{3}$$

where zero-padding fills gaps in the sequence.

1.2 Adjustments for Transformer Processing

To make YOLO outputs compatible with Transformer models, dimensionality reduction, embedding, and positional encoding are required.

1.2.1 Dimensionality Reduction and Embedding

Each bounding box data is embedded into a fixed-size vector:

$$\mathbf{e}_t = f_{\text{embed}}(\mathbf{B}_t) \tag{4}$$

where f_{embed} is the embedding function, mapping bounding box coordinates to a vector space compatible with the Transformer's input requirements.

1.2.2 Positional Encoding

Temporal information is crucial for behavior analysis. Transformers incorporate positional encodings, calculated as follows:

$$\text{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right) \tag{5}$$

$$\text{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right) \tag{6}$$

where pos is the position in the sequence, i the dimension, and d the embedding dimension. This provides a sense of temporal order.

1.3 Technical Trade-offs: Speed, Real-Time Constraints, and Accuracy

1.3.1 Speed and Real-Time Constraints

YOLO operates in real-time, yet Transformers, especially for long sequences, can face computational bottlenecks. Efficient attention mechanisms (e.g., Linformer) reduce Transformer memory complexity from $O(T^2)$ to $O(T)$.

1.3.2 Sequential Length and Memory Use

For long behavior sequences, we divide sequences into overlapping chunks:

$$\mathbf{S}_{\text{chunks}} = \{\mathbf{S}_{1:k}, \mathbf{S}_{k+1:2k}, \dots\} \tag{7}$$

where k is the chunk size. This maintains local context while reducing memory use.

1.4 Behavior Prediction and Contextual Analysis

1.4.1 Object Movement and Temporal Changes

Behavior analysis often requires detecting movement and changes over time. Given a bounding box sequence for an object, the velocity at frame t can be approximated by:

$$\mathbf{v}_t = \frac{\mathbf{B}_t - \mathbf{B}_{t-1}}{\Delta t} \tag{8}$$

where Δt is the frame interval. Velocity vectors provide additional input to the Transformer, improving its ability to model dynamic behaviors.

1.4.2 Handling Noise and False Positives

YOLO can output noisy predictions, affecting Transformer performance. A Kalman filter is often applied for smoothing:

$$\hat{\mathbf{B}}_t = K_t\mathbf{B}_t + (1 - K_t)\hat{\mathbf{B}}_{t-1} \tag{9}$$

where $\hat{\mathbf{B}}_t$ is the estimated bounding box and K_t the Kalman gain. This produces smoother sequences for the Transformer.

1.5 Proposed Architecture and Workflow

The proposed architecture includes:

1. **YOLO for Object Detection and Initial Tracking:** YOLO detects objects in each frame, outputting bounding boxes.
2. **Preprocessing for Transformer:** YOLO outputs are embedded, and positional encoding is applied.
3. **Transformer for Behavior Analysis:** The Transformer processes bounding box sequences to identify behavior patterns.

2 Evaluating the Effectiveness of Using Transformers for Temporal Behavior Prediction

Transformer models are highly effective in predicting the behavior of detected objects in video streams by analyzing the temporal sequence of YOLO's detection outputs. Their self-attention mechanism offers significant advantages over traditional RNN-based models for understanding movement patterns over time, particularly in scenarios such as detecting suspicious activities in surveillance footage or classifying vehicle speed and direction in traffic.

2.1 The Self-Attention Mechanism in Transformers

The self-attention mechanism is the core of Transformer models and provides unique benefits in temporal behavior prediction:

- **Capturing Long-Range Dependencies**: Transformers analyze all frames within a sequence simultaneously, capturing dependencies across long time intervals. In video analysis, this capability is crucial since events unfold over multiple frames, requiring long-term context for accurate behavior prediction.
- **Selective Attention to Relevant Frames**: Self-attention enables the model to prioritize frames with the most informative details. For example, in traffic monitoring, attention can be drawn to frames where a vehicle changes lanes or accelerates, focusing the model on significant movements.
- **Flexibility in Sequence Length**: Unlike RNNs, which process frames sequentially, Transformers handle entire sequences in parallel, allowing them to analyze frames without being restricted by order, which is particularly advantageous in real-time applications.

In contrast, RNN-based models (e.g., LSTMs, GRUs) are sequential in nature, which often makes them slower and limits their ability to capture long-term dependencies. They suffer from vanishing gradients and limited memory, making them less suited to long-sequence analysis compared to the Transformer's self-attention mechanism.

2.2 Mathematical Foundations of Transformer-Based Behavior Prediction

To further understand the technical effectiveness of Transformers in behavior prediction, here are the relevant mathematical formulations:

2.2.1 Self-Attention Mechanism

Each input frame $\mathbf{X}_t$ is transformed into Query (Q), Key (K), and Value (V) vectors:

$$Q = \mathbf{X}W_Q, \quad K = \mathbf{X}W_K, \quad V = \mathbf{X}W_V \tag{10}$$

where W_Q, W_K, and W_V are learned weight matrices. The attention score between frames i and j is calculated as:

$$\text{Attention}(Q_i, K_j) = \frac{\exp(Q_i K_j^T / \sqrt{d_k})}{\sum_{j=1}^{T} \exp(Q_i K_j^T / \sqrt{d_k})} \tag{11}$$

This score represents the importance of each frame relative to others, allowing the model to focus on frames that carry critical behavioral information.

2.2.2 Positional Encoding for Temporal Order

To add temporal order to frames, Transformers use positional encodings. For each frame position *pos* and dimension i, these encodings are defined as:

$$\text{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right) \tag{12}$$

$$\text{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right) \tag{13}$$

These values are added to each input embedding, giving the model an awareness of temporal progression within the sequence.

2.2.3 Velocity and Acceleration from YOLO's Bounding Boxes

By extracting velocity and acceleration as additional features from YOLO's bounding box detections, we can improve the Transformer's ability to recognize movement patterns:

- **Velocity**: Estimated for an object in consecutive frames t and $t+1$ as:

$$\mathbf{v}_t = \frac{\mathbf{B}_{t+1} - \mathbf{B}_t}{\Delta t} \tag{14}$$

- **Acceleration**: Calculated as:

$$\mathbf{a}_t = \frac{\mathbf{v}_{t+1} - \mathbf{v}_t}{\Delta t} \tag{15}$$

where $\mathbf{B}_t$ represents bounding box coordinates at frame t, and Δt is the time interval. These features help capture sudden behavioral shifts, such as acceleration that may indicate erratic or suspicious movement.

2.3 Scenario Analysis: Applications of Transformer-Based Behavior Prediction

2.3.1 Scenario 1: Detecting Suspicious Activities in Surveillance

Transformers are well-suited for identifying suspicious activities, as they can recognize subtle, long-term patterns in movement:

- **Abnormal Movements**: Anomalies like pacing or loitering are detectable by tracking shifts in position and speed across frames. The self-attention mechanism highlights significant frames where these behaviors occur, allowing Transformers to detect and flag such activities as unusual.
- **Predictive Behavior Analysis**: In situations where sudden actions (like a person moving rapidly toward a restricted area) indicate potential threats, Transformers can predict these behaviors based on prior frames.

2.3.2 Scenario 2: Traffic Analysis for Speed and Direction Classification

In traffic monitoring, Transformers excel in classifying vehicle behaviors:

- **Speed and Direction Prediction**: By analyzing bounding box sequences, Transformers can detect changes in speed and direction, helping classify events like lane changes or erratic driving.
- **Traffic Violation Detection**: Transformers are capable of tracking vehicles across extended sequences, flagging possible violations such as speeding or unauthorized lane changes.

2.4 Visualizations and Graphs to Enhance Understanding

To illustrate how Transformers predict temporal behavior, here are key visualizations:

- **Attention Heatmap**: This heatmap visualizes frame-to-frame attention scores, showing which frames the Transformer focuses on for behavior prediction. For instance, a heatmap for vehicle monitoring may show high attention on frames where the vehicle changes speed or direction.
- **Velocity and Acceleration Over Time**: Plotting velocity and acceleration over frames reveals dynamic changes, such as sudden stops or accelerations, which might indicate unusual behavior.
- **Comparison Graphs for Transformer vs. RNN**:
  - **Accuracy vs. Sequence Length**: This line graph demonstrates how Transformer accuracy remains high over longer sequences, while RNN accuracy decreases.
  - **Real-Time Processing Speed**: A bar chart comparing processing times highlights Transformer efficiency in handling larger sequences more accurately, despite being computationally intensive.
- **Trajectory Prediction for Suspicious Activity**: Plotting actual vs. predicted trajectories shows the Transformer's ability to anticipate movements based on observed behaviors. Peaks in anomaly scores indicate suspicious patterns such as pacing or sudden direction changes.

2.5 Performance Metrics for Transformer-Based Behavior Prediction

Quantitative metrics for evaluating Transformer models in behavior prediction include:

- **Accuracy**: Measures predictive accuracy for future bounding box locations or behavioral classifications.
- **Precision and Recall in Anomaly Detection**: Essential for scenarios like surveillance, where distinguishing abnormal from normal behavior is critical.
- **Mean Absolute Error (MAE) in Trajectory Prediction**: This metric evaluates the prediction accuracy for bounding box coordinates, vital for applications requiring precise movement tracking.
- **Model Latency and Frame Processing Rate**: Evaluates real-time feasibility by measuring processing time per frame, essential for practical applications like surveillance.

3 Impact of Model Optimization on Real-Time Performance in YOLO-Transformer Systems

Model optimization is crucial when combining YOLO and Transformer architectures for real-time video analysis, especially in scenarios where computational resources are limited, such as on mobile devices or edge computing platforms. Optimizations like model quantization, input size reduction, and parameter pruning enable faster processing but introduce trade-offs between detection accuracy and the temporal analysis quality of Transformers. Below, we explore these optimizations in detail, accompanied by relevant equations and visualizations.

3.1 Key Model Optimization Techniques

3.1.1 Model Quantization

Quantization reduces the precision of model weights and activations, often from 32-bit floating-point (FP32) to lower precisions like 8-bit integers (INT8). This results in faster computation and reduced memory usage at the potential cost of accuracy.

- **Quantized Weight Representation**: Let W be a weight matrix in FP32 format. Quantization maps each weight $w \in W$ to a lower bit representation $\hat{w}$ using:

$$\hat{w} = \text{round}\left(\frac{w - \min(W)}{\text{scale}}\right) \cdot \text{scale} + \min(W)$$

where scale $= \frac{\max(W)-\min(W)}{2^b-1}$ and b is the bit width (e.g., $b = 8$ for INT8). This reduces model size, making it feasible for low-memory devices, though some precision loss is introduced.

- **Effect on YOLO and Transformers**:
  - *YOLO*: Faster processing of detection outputs, allowing real-time performance. However, quantization can lead to small inaccuracies in bounding box coordinates.
  - *Transformers*: With INT8 quantization, Transformers benefit from reduced memory usage and faster matrix multiplications, especially in the self-attention layers, at a slight cost to temporal precision.

3.1.2 Input Size Reduction

Reducing the input resolution minimizes the pixel count, resulting in faster processing times but potentially losing detail, especially for small objects.

- **Effect on Bounding Box Predictions**: With reduced input sizes, YOLO may yield slightly coarser bounding boxes. If the input size is scaled by a factor s, then for a given bounding box $B = (x, y, w, h)$, where x and y are center coordinates and w, h are width and height, the adjusted bounding box is:

$$B' = (sx, sy, sw, sh)$$

- **Impact on Temporal Analysis**:
  - Reduced input size affects the detail of YOLO's bounding box outputs, potentially impacting the Transformer's ability to perform fine-grained temporal analysis, as small, gradual changes in motion may become less discernible.

3.1.3 Parameter Pruning

Pruning removes weights or units deemed less critical, reducing the model's size and speeding up inference.

- **Pruning Mechanism**: Weights with magnitudes below a certain threshold θ are set to zero. Given a weight matrix W, pruning can be represented as:

$$W_{\text{pruned}} = \{w \in W \mid |w| > \theta\}$$

- **Effect on YOLO and Transformers**:
  - *YOLO*: Pruning can reduce feature extraction complexity, which speeds up detection but may cause slight accuracy degradation if important features are pruned.
  - *Transformers*: In Transformers, pruning self-attention layers or heads may speed up inference but can impact the model's ability to capture temporal dependencies.

3.1.4 Knowledge Distillation

Knowledge distillation trains a smaller "student" model to approximate the performance of a larger "teacher" model, retaining much of the larger model's knowledge with fewer parameters.

- **Distillation Loss**: Knowledge distillation involves minimizing a combined loss function:

$$L = \alpha \cdot L_{\text{CE}}(y, \hat{y}) + (1 - \alpha) \cdot L_{\text{KD}}(z, \hat{z})$$

  where L_{CE} is the cross-entropy loss for true labels y and predictions $\hat{y}$, and L_{KD} is the knowledge distillation loss for the teacher logits z and student logits $\hat{z}$. The parameter α controls the balance between true labels and teacher guidance.

- **Effect on YOLO and Transformers**:

- A distilled YOLO model retains much of the detection quality with a smaller architecture, making it more suitable for edge devices.
- A distilled Transformer can approximate the temporal analysis abilities of the teacher model with reduced computational requirements, though it may sacrifice some precision in predicting fine-grained temporal details.

3.2 Visualizations and Graphs for Optimization Effects

- **Quantization Impact on Bounding Box Accuracy**:
  - *Scatter Plot*: Visualize the difference between original bounding box coordinates and quantized bounding box coordinates to observe precision loss.
  - *Bar Chart*: Compare detection accuracy (e.g., mAP) of the original vs. quantized models, showing the trade-off between reduced memory and accuracy.
- **Detection Speed vs. Input Size**:
  - *Line Graph*: Plot detection time per frame against input size to show the speed gains from input reduction.
  - *Object Size Detection Accuracy*: A bar chart can show how input reduction affects detection accuracy for small, medium, and large objects.
- **Memory Usage vs. Pruning Percentage**:
  - *Memory Graph*: Plot memory usage as a function of pruning percentage, showing the trade-off between memory savings and model accuracy.
  - *Performance Decay Curve*: A line graph showing how accuracy decreases with increasing pruning percentage helps highlight an optimal pruning threshold.
- **Accuracy and Latency Comparison for Distilled Models**:
  - *Line Graphs*: Compare latency and accuracy of the original and distilled models, demonstrating the distillation's effect on performance trade-offs.

3.3 Quantitative Impact on Detection Accuracy and Temporal Analysis

To illustrate the quantitative impact of optimization, specific metrics are used:

- **Mean Average Precision (mAP)**:

$$\text{mAP} = \frac{1}{|O|} \sum_{o \in O} \text{AP}(o)$$

  where $\text{AP}(o)$ is the average precision for object o. mAP can be calculated for both the original and quantized YOLO models to assess accuracy loss due to quantization.

- **Mean Absolute Error (MAE) in Temporal Analysis**:

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^{N} \left| \hat{B}_i - B_i \right|$$

  where $\hat{B}_i$ and B_i are the predicted and actual bounding box coordinates, respectively. Higher MAE suggests more loss in temporal precision due to quantization or input reduction.

- **Inference Latency (ms/frame)**: Measuring latency in milliseconds per frame is essential for evaluating real-time suitability. Latency can be calculated for each optimization approach (quantized, pruned, reduced input) and compared to the baseline.

- **Frames Per Second (FPS)**: FPS measures real-time capability. Optimized models typically have higher FPS, allowing for smoother video analysis. FPS values can be plotted for various optimization levels to illustrate trade-offs between speed and accuracy.

3.4 Trade-Off Analysis in Real-Time YOLO-Transformer Systems

The following table summarizes the trade-offs in terms of model size, detection accuracy, and temporal analysis quality for each optimization technique:

Optimization Technique	Model Size Reduction	Detection Accuracy Impact	Temporal Analysis Accuracy	Latency Improvement
Quantization	High	Moderate (small loss)	Moderate	High
Input Size Reduction	High	High (affects small objects)	Moderate to High	Very High
Parameter Pruning	Medium	Low to Moderate	Moderate	Medium
Knowledge Distillation	Medium	Low	Low to Moderate	Medium

4 Optimizing BERT for Large-Scale and Real-Time Tasks

Optimizing BERT for large-scale or real-time tasks is crucial for reducing computational overhead and maintaining efficiency, especially for complex applications like sequence analysis. Here, we explore strategies for optimizing BERT, including GPU acceleration, reducing input length, batching, model distillation, quantization, and more.

4.1 Leveraging GPU Acceleration in Google Colab

Using a GPU can dramatically speed up BERT's computation time for both training and inference. To enable GPU acceleration in Google Colab:

- Go to **Runtime ¿ Change runtime type ¿ Hardware accelerator** and select **GPU**.
- In your code, confirm GPU availability and move your model and data to the GPU:

```
import torch
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

Running on a GPU provides significant speed improvements for handling large datasets and performing real-time tasks with BERT.

4.2 Reducing Input Sequence Length

BERT's processing time grows quadratically with the input sequence length due to its self-attention mechanism. Reducing sequence length is one of the most effective ways to improve performance:

- **Trim Inputs**: Retain only the essential part of each input text. For example, for classification tasks, setting the maximum sequence length to **128 tokens** is often sufficient.
- **Truncation and Padding**: Use BERT's tokenizer parameters `truncation` and `padding` to keep all inputs consistent:

```
inputs = tokenizer(text, return_tensors="pt", padding="max_length",
                   truncation=True, max_length=128)
```

Reducing the sequence length is particularly useful for tasks where context beyond a certain length does not add significant value.

4.3 Batching Inputs for Efficiency

Batching allows multiple inputs to be processed in parallel, maximizing GPU utilization and reducing processing time:

- **Batch Size**: Choose a batch size that maximizes GPU utilization without exhausting memory. Batch sizes like **8, 16, or 32** are common, but experiment to find the optimal size for your setup.
- **Dataloader**: Use a PyTorch `DataLoader` to manage batching efficiently:

```
from torch.utils.data import DataLoader
train_dataloader = DataLoader(train_dataset, batch_size=16, shuffle=True)
```

Batching improves computational efficiency, especially when handling large datasets or real-time data streams.

4.4 Model Optimizations: Distillation and Quantization

4.4.1 Distillation

Model distillation is a technique where a smaller "student" model learns from a larger, pre-trained "teacher" model. This process reduces the model size while preserving much of BERT's accuracy:

- **DistilBERT**: A distilled version of BERT that is about **60% smaller** and nearly as accurate. It's an excellent option for real-time and mobile applications.

```
from transformers import DistilBertTokenizer, DistilBertForSequenceClassification
tokenizer = DistilBertTokenizer.from_pretrained("distilbert-base-uncased")
model = DistilBertForSequenceClassification.from_pretrained("distilbert-base-uncased")
```

4.4.2 Quantization

Quantization reduces model size and speeds up inference by converting model weights from 32-bit floating-point to lower precision, such as 8-bit integers:

- **Dynamic Quantization**: This approach is useful for reducing model size, especially when deploying models to CPU-based environments. Here's an example:

```
from transformers import BertForSequenceClassification
model = BertForSequenceClassification.from_pretrained("bert-base-uncased")
model = torch.quantization.quantize_dynamic(model, {torch.nn.Linear}, dtype=torch.qint8)
```

Quantization is especially beneficial for deploying models on edge devices or mobile platforms with limited computational resources.

4.5 Gradient Checkpointing

Gradient checkpointing saves memory by only storing intermediate activations when needed during backpropagation. This allows for larger batch sizes or longer sequences without exceeding memory limits:

- **Enable Gradient Checkpointing**: In Hugging Face's `Trainer`, set `gradient_checkpointing=True` in `TrainingArguments`:

```
from transformers import TrainingArguments
training_args = TrainingArguments(
    output_dir="./results",
    gradient_checkpointing=True,
    per_device_train_batch_size=8,
    num_train_epochs=3,
)
```

This can save 30-50% of memory, which is highly useful for tasks involving long sequences or large batch sizes.

4.6 Mixed Precision Training

Mixed precision training reduces memory usage and accelerates computation by using lower-precision formats (like FP16) where possible. This approach, known as **Automatic Mixed Precision (AMP)**, is supported on most NVIDIA GPUs:

```
from transformers import Trainer, TrainingArguments

training_args = TrainingArguments(
    output_dir="./results",
    per_device_train_batch_size=16,
    fp16=True  # Enables mixed precision
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset
)
trainer.train()
```

Mixed precision can double the training and inference speed on compatible hardware, making it ideal for large-scale tasks.

4.7 Data Pipeline Optimization

Efficient data handling is essential for real-time applications, as data processing can become a bottleneck:

- **Preprocess Data in Advance**: Tokenize and preprocess data beforehand to reduce runtime delays.
- **Asynchronous Data Loading**: Use multiple workers in `DataLoader` to load data asynchronously:

  ```
  train_dataloader = DataLoader(train_dataset, batch_size=16, shuffle=True, num_workers=4)
  ```

By streamlining data loading, you ensure that BERT can access data as quickly as possible, reducing idle time on the GPU.

4.8 Summary of Optimization Techniques

To optimize BERT for large-scale or real-time tasks:

- **GPU Acceleration**: Enable GPU for faster processing in Google Colab or other environments.
- **Reduce Input Sequence Length**: Truncate inputs to focus on essential content.
- **Batch Inputs**: Use batch processing for parallel computation.
- **Model Optimizations**: Leverage distillation (e.g., DistilBERT) and quantization for smaller, faster models.
- **Gradient Checkpointing and Mixed Precision**: Use these techniques to save memory and accelerate training.
- **Optimize Data Pipeline**: Preprocess and load data asynchronously to minimize runtime delays.

Combining these techniques can significantly enhance BERT's performance, making it suitable for deployment in large-scale and real-time applications. This setup maximizes efficiency while retaining high accuracy, making BERT adaptable to both complex and resource-constrained environments.

5 Technical Analysis Report on BERT Fine-Tuning and Prediction Workflow

This report covers the workflow for fine-tuning a BERT model on a custom dataset for action recognition and making predictions. Each section corresponds to a specific stage of the process.

5.1 Loading and Tokenizing Input Text

1. **Code Overview**:
   - The code begins by importing necessary modules: `BertTokenizer`, `BertForSequenceClassification`, and `torch`.
   - A pre-trained BERT model (`bert-base-uncased`) and its tokenizer are loaded.
   - A sample text input, `"start move_right move_right accelerate move_left stationary stop"`, is tokenized using BERT's tokenizer.
2. **Execution**:
   - The code tokenizes the input and generates `input_ids` (token IDs) and an `attention_mask`.
   - `input_ids`: Unique IDs corresponding to tokens in the input text.
   - `attention_mask`: Binary mask indicating which tokens should be attended to by the model (1 for real tokens, 0 for padding).
3. **Output**:
   - The model processes the tokenized input, and the predicted class ID is outputted. Here, the predicted class is `1`, which corresponds to "running".

5.2 Mapping Predicted Output to Actions

1. **Action Mapping**:
   - The predicted class ID is mapped to an action. In this example, the actions are defined as:
     - `0`: "walking"
     - `1`: "running"
     - `2`: "stationary"
2. **Output**:
   - For the input sequence, the predicted action is `"running"`.

5.3 Dataset Creation

1. **Dataset Class Definition**:
   - A custom dataset class, `ActionRecognitionDataset`, is created to handle the input text sequences and labels.
   - The dataset class takes in `texts` (list of action sequences) and `labels` (corresponding action categories) and tokenizes each sequence.
2. **Sample Dataset**:
   - A small dataset with sample sequences and corresponding labels is created:
     - Texts represent different movement patterns.
     - Labels are integers (0 for "walking," 1 for "running," and 2 for "stationary").
3. **Dataloader Creation**:
   - The dataset is loaded into a PyTorch `DataLoader` for batching and efficient processing.

5.4 Loading the Model for Sequence Classification

1. **Model Setup**:
   - A BERT model for sequence classification (`BertForSequenceClassification`) is initialized with 3 output labels.
   - A warning appears indicating that some weights (classification layer weights) are newly initialized because this is a pre-trained BERT model with an added classification head for custom labels.

5.5 Training Configuration and Initialization

1. **Training Arguments**:
   - The `TrainingArguments` object specifies key parameters for training:
     - `output_dir`: Directory to save model outputs.
     - `evaluation_strategy`: Set to "no" for training without validation.
     - `learning_rate`: Set to `2e-5`, which is a common fine-tuning rate for BERT.
     - `per_device_train_batch_size`: Set to `2` due to memory constraints.
     - `num_train_epochs`: Set to `3` for a brief training run.
     - `weight_decay`: Regularization parameter to prevent overfitting.
2. **Trainer Initialization**:
   - A `Trainer` object is created, which automates the training loop with the specified arguments.
3. **Warnings**:
   - A `FutureWarning` for the `evaluation_strategy` parameter indicates that it will be replaced by `eval_strategy` in a future version of the library.

5.6 Training Process and W&B Integration

1. **Training Start**:
   - The training process is started using the `trainer.train()` method.
2. **Weights & Biases (W&B) Logging**:
   - The code integrates with W&B for experiment tracking. A warning appears indicating that the `run_name` is set to the same value as `output_dir`.
   - Instructions for logging into W&B are provided. W&B tracks metrics such as `train_runtime`, `train_samples_per_second`, and `train_loss` for each epoch.
3. **Training Output**:
   - After 3 epochs, the final training loss is reported as `1.1166`, which provides a quick measure of the model's training performance.
4. **W&B Deactivation**:
   - An optional code snippet is shown to disable W&B logging by setting `report_to="none"` in `TrainingArguments`.

5.7 Prediction Function and Final Prediction

1. **Prediction Function Definition**:
   - A function `predict` is defined to make predictions on new text sequences.
   - This function tokenizes the input, moves it to the appropriate device, runs it through the model, and retrieves the predicted class.
2. **Mapping and Output**:
   - The predicted class is mapped to an action (e.g., "walking," "running," or "stationary").
   - For the test input `"start move_forward move_forward stop"`, the predicted action is `"running"`.

5.8 Summary

This workflow demonstrates the entire process of fine-tuning and using BERT for a custom sequence classification task in action recognition. Key steps include:

- Loading and tokenizing data.
- Creating a custom dataset and dataloader.
- Initializing a pre-trained BERT model with a new classification head.
- Configuring training parameters and running the fine-tuning process.
- Implementing a prediction function for new input sequences.

This setup can be adapted for real-time action recognition or similar tasks requiring custom classifications based on sequence data. The report also provides options for managing experiment logging, specifically with Weights & Biases (W&B).