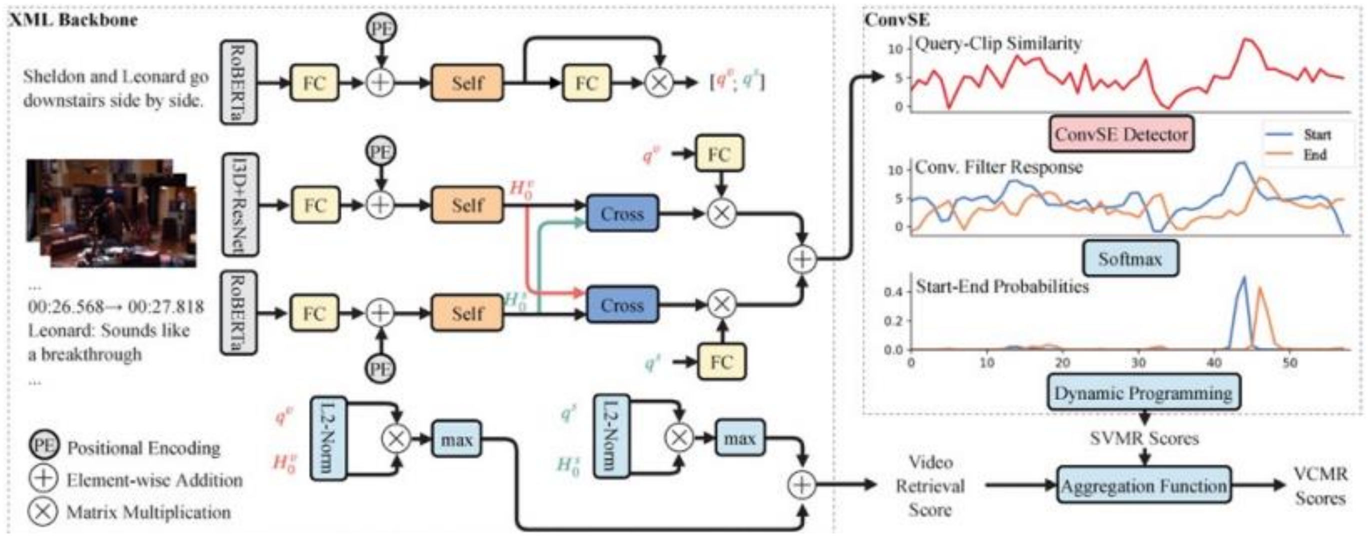
Project 3 Report

Name: Pallavi Das

This report is based on the project of implementing a video moment retrieval. The video moment retrieval defines the localizing of a specific time span in a video from query. For the content, the pipeline and its purpose has been described as well as the reason behind why XML backbone has been developed.

In this report, the workflow or pipeline would be described. It will be done utilizing the given python file: **model_xml.py**



This is the schematic of the pipeline utilized by Single Video Moment Retrieval (SVMR).

Description of the XML Backbone purpose:

SVMR (Single Video Moment Retrieval) is a current state-of-the-art technique for retrieval of video data. Its goal is to achieve successful retrieval of a moment from a single video with the help of a natural language query. There has also been modification in SVMR with extensions to Video Corpus Moment Retrieval (VCMR). VCMR is also a system for video moment retrieval with its base as SVMR. This system requires to retrieve the most essential and relevant moments from a set of videos instead of just a single video in SVMR.

This method or system utilizes the visual modality as the source of content for retrieval process as most of the existing related datasets currently are based on the video content system. In other words, it can be stated that SVMR is a simplified version of VCMR.

However, the existing current state-of-the-art method consists of some problems affecting the efficiency of the goal of the system. As these methods utilize handcrafted heuristics/assumptions, or sliding window methods, hence they lack precision when it comes to temporal space. This only limits the goal performance to be achieved with suboptimal efficiency.

To address these above-mentioned challenges and many more, Cross-modal Moment Localization (also known as XML) has been developed. It's an approach incorporated with the VCMR method.

Some of the features embedded in XML are as follows:

1. XML consists of videos and/or subtitles and queries that are encoded independently. So, for a system of retrieving Y queries from a set of X videos, only X+Y operations of neural networks would be needed to complete this operation.
2. In XML system, videos can be encoded beforehand and then stored. For testing purposes, and user would only need to work with newly set queries. This aids in reducing the waiting time for encoding newly requested user queries.
3. Core element of XML approach is the **ConvSE (Convolutional Start-End detector)** module. This module learns to detect or predict the start to end edges in a one-dimensional signal along with two convolutional filters.
   - ⇨ **ConvSE**: It's a detector module that detects the up and down edges in the similar signals that consists of two trainable one-dimensional convolutional filters. This provides a much better performance especially in terms of computational efficiency. In other words, up and down edges refers to the start and end edges.

XML Backbone Model/Module:

1. Input Representations/Initialization for the module:
   For representing videos, it is required to consider both the appearance and motion features of the given video dataset. The given data in the XML backbone schematic contains video features, that is, ResNet, 13D as well as text features, that is, subtitle and query obtained from fine-tuned RoBERTa. For display/appearance, the clip-level feature is obtained, while for motion, 13D features are extracted. These two features are then concatenated using L2-normalization. Contextualized text features are extracted using RoBERTa of 12-layers. As mentioned earlier, subtitles and queries are obtained by fine tuning RoBERTa. The parameters associated with this are fixed to extract contextualized token embeddings.
   Queries directly use the token embeddings that have been extracted.
   In case of subtitles, the token-level embeddings are first extracted, then max pool is applied to obtain clip-level feature vector.
   The dimensions of these extracted feature vectors are reduced into a low-dimensional space with the help of a linear layer along with ReLU activation function. These features are then projected by embedding the learned positional encodings.

2. Query Encoding Component:
   As we already know, a TVR query can be categorized into two labels, namely video or subtitle. In implementation, the query can be divided, that too, dynamically, into two modularized vectors. The Self-Encoder is then used for encoding a query feature vector. The Self-Encoder contains s4 main components, which are self-attention layer, linear layer, along with a residual connection, then a layer normalization. This information is then used to calculate the attention scores for each query corresponding to the video segment or subtitle segment. Then, modularized query vectors are generated by aggregating the attention scores.

The following code snippet is used for obtaining the modularized query vector values that is computed by aggregating the attention scores for each respective video and/or subtitle segments.

```
def encode_query(self, query_feat, query_mask):
    # TODO
    # ========================================
    # Use self.encode_input to encode the query
    encoded_query = None  # (N, Lq, D)
    # Modularize queries
    video_query, sub_query = None  # (N, D) * 2
    # ========================================
    return video_query, sub_query

def non_cross_encode_context(self, context_feat, context_mask, module_name="video"):
    encoder_layer3 = getattr(self, module_name + "_encoder3") \
        if self.config.encoder_type == "transformer" else None
    return self._non_cross_encode_context(context_feat, context_mask,
                                          input_proj_layer=getattr(self, module_name + "_input_proj"),
                                          encoder_layer1=getattr(self, module_name + "_encoder1"),
                                          encoder_layer2=getattr(self, module_name + "_encoder2"),
                                          encoder_layer3=encoder_layer3)
```

3. <u>Context Encoding Component:</u>

Two Self encoders are used on the provided features of video and subtitle in the pipeline. These self-encoders are used to compute their corresponding single-model contextualized features. These cross-modal representations are then encoded using the Cross Encoder. The Cross encoder takes the self-modality and cross modality features ad inputs, which is further encoded utilizing cross-attention. Next the pipeline continues by sending the processed encoded information to a linear layer, then a residual connection, followed by a layer normalization and finally a third self-encoder.

The following code snippet computes the single-model contextualized features using the given video and subtitle features and passing it as inputs to two self-encoders respectively.

```
def encode_context(self, video_feat, video_mask, sub_feat, sub_mask):
    if self.config.cross_att:
        assert self.use_video and self.use_sub
        return self.cross_encode_context(video_feat, video_mask, sub_feat, sub_mask)
    else:
        video_feat1, video_feat2 = (None,) * 2
        if self.use_video:
            video_feat1, video_feat2 = self.non_cross_encode_context(video_feat, video_mask, module_name="video")
        sub_feat1, sub_feat2 = (None,) * 2
        if self.use_sub:
            sub_feat1, sub_feat2 = self.non_cross_encode_context(sub_feat, sub_mask, module_name="sub")
        return video_feat1, video_feat2, sub_feat1, sub_feat2
```

The following code snippets is used for encoding the cross-modal representations obtained from above code snippet. The encoding is done using the Cross Encoder.

```
def cross_encode_context(self, video_feat, video_mask, sub_feat, sub_mask):
    # TODO
    # ========================================
    # Encode video features
    encoded_video_feat = None
    # Encode subtitle features
    encoded_sub_feat = None
    # Cross encode video with subtitle features
    x_encoded_video_feat = None  # (N, L, D)
    # Cross encode subtitle with video features
    x_encoded_sub_feat = None  # (N, L, D)
    # ========================================
    return encoded_video_feat, x_encoded_video_feat, encoded_sub_feat, x_encoded_sub_feat
```

```
def cross_context_encoder(self, main_context_feat, main_context_mask, side_context_feat, side_context_mask,
                          cross_att_layer, norm_layer, self_att_layer):
    """
    Args:
        main_context_feat: (N, Lq, D)
        main_context_mask: (N, Lq)
        side_context_feat: (N, Lk, D)
        side_context_mask: (N, Lk)
        cross_att_layer:
        norm_layer:
        self_att_layer:
    """
    cross_mask = torch.einsum("bm,bn->bmn", main_context_mask, side_context_mask)  # (N, Lq, Lk)
    cross_out = cross_att_layer(main_context_feat, side_context_feat, side_context_feat, cross_mask)  # (N, Lq, D)
    residual_out = norm_layer(cross_out + main_context_feat)
    if self.config.encoder_type in ["cnn", "transformer"]:
        return self_att_layer(residual_out, main_context_mask.unsqueeze(1))
    elif self.config.encoder_type in ["gru", "lstm"]:
        return self_att_layer(residual_out, main_context_mask.sum(1).long())[0]
```

Convolutional Start-End (ConvSE) Detector:

- From the XML Backbone, the final video and subtitle representations from Context Encoding component and modularized query vector values for video and subtitle representations from Query Encoding component are obtained. Now, these obtained values are used for computing the query-clip similarity scores within this ConvSE Detector.
- ConvSE is a detector mechanism or module accompanied with two one-dimensional convolution filters. It helps to detect the start and end edges (that is, the up and down edges) in the query-clip similarity scores curve. It makes use of two trainable filters (without any bias) in order to generate the starting and ending scores.
- The starting and ending score values are then normalized with the help of softmax function to produce the resulting probability values.

The code snippet for the ConvSe module dataset has been provided in the hw3 code set under the python file name of start_end_dataset.py file.

References/Resources:

-Class notes

-Given code in Homework3

-TVR paper analysis