Video Description

Zeibel Antonia

1 Introduction

The aim of the project is to describe the input video using meaningful sentences. The project relies on a combination between Convolutional Neural Networks for feature extraction, and Recurrent Neural Networks to capture the temporal dependencies between the frames, also known as Long-term Recurrent Convolutional Network (LRCN). For the project, I will review some of the articles on this specific subject, which are presented in the [1].

2 Methodology

2.1 Input Video Processing

First of all, the frames contained in the video are defined as: key-frames and in-between frames. The key-frames are frames which present relevant information for the captioning task, and the in-between frames are just the frames between such key-frames. Basically, the key-frames can be extracted using uniform sampling (select frames at fixed intervals of time) or scene change detection techniques. To maintain the temporal dependencies between the key-frames, the in-between frames are selected as well. The reasoning for this processing step is to reduce the computational overhead.

Other processing steps can be considered in this stage, like resizing the frames to a consistent resolution, and normalizing pixel values for the CNN.

2.2 Video Encoder (Convolutional Neural Network)

The next stage of the pipeline is the visual feature extraction performed on each frame in the input using CNN. Each CNN outputs a feature vector corresponding to that specific frame. By the end of this stage, all these feature vectors are represented by a final sequence of feature vectors that respects the temporal constraints of the frames, and also contain spatial features. This sequence of vectors serves as input for the next stage of the pipeline, video decoding, combining both spatial and temporal features.

2.3 Video Decoder (Recurrent Neural Network)

The encoded visual features are fed to the RNN to generate the actual captioning of the video. As for the RNN architecture, I will choose a Long Short-Term Memory (LSTM) network. LSTM networks are considered a solution for sequence prediction tasks because of their ability to capture long-term dependencies.

It is important to note the presence of two special tokens: **<BOS>** and **<EOS>**.

- **<BOS>** which means that the system finds itself to the beginning of the sequence.
- **<EOS>** which marks the end of the sequence.

The decoder is initialized using the encoded visual features and generates words one at a time, starting from <BOS>. Each output from that moment on is then used for the next step, until the <EOS>. As a result, the LSTM decoder generates the video description consisting of all the generated words at each time step.

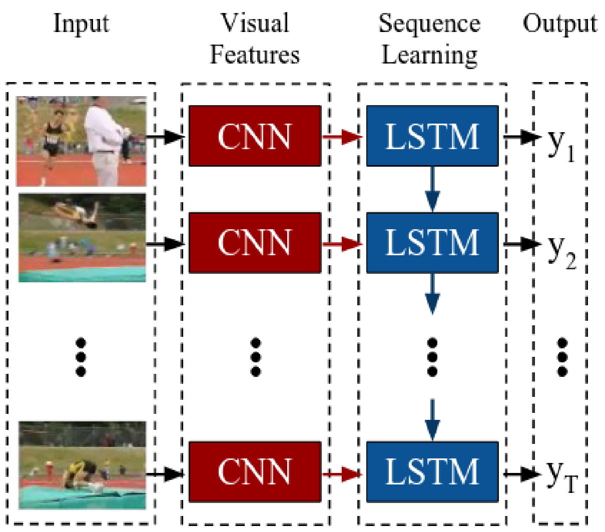


Figure 1: LRCN architecture as proposed in [2]

3 Data

The data used for this captioning task was the Microsoft Research Video Description Corpus (MSVD) [3], primarily used for video description or video captioning tasks in machine learning and natural language processing, and also the video snippets themselves [4].

4 Implementation

The actual implementation is done within **four main stages**, translated into notebooks:

1. `video_frame_extraction.ipynb`
2. `frame_preprocessing.ipynb`
3. `captions_preprocessing.ipynb`
4. `caption_generator.ipynb`

4.1 Video Frame Extraction

This first stage implies reading the video_corpus.csv file containing the annotations, and other information regarding the video files: VideoID, Start, End, WorkerID, Source, AnnotationTime, Language, and Description.

I was only interested in the English annotations, therefore I dropped all the other rows having a different language, other than English. Besides these non-English rows, I also dropped some of the columns that were irrelevant to the task itself, like AnnotationTime, Language and WorkerID.

After these clean-up steps, I decided to double-check if the video files actually have a correspondent into the video corpus (check for annotations). Of course, this step was applied to all the video files in the directory. If there was such video, I would just save it into a list with video files with no annotations associated.

Now the video files were ready for frame extraction. I have implemented two methods which have this role: `frame_extraction(video_path)` and `extract_frames(video_dir)`. Basically, the first method mentioned just transforms the video into a list of frames, and saves them into their specific directory created using their unique VideoID. The second method actually iterates the video files, and passes to the first method mentioned the right paths. I also have two separate list with the IDs of extracted video files, and the IDs of the failed to extract video files.

4.2 Frame Selection

After successfully extracting the frames from the video files, I have implemented multiple techniques to choose the frames for the next stage of the pipeline. Being realistic, video files do not have a fixed number of frames, and different frames in the video might actually be more informative than the others. Moreover, based on the architecture of the employed machine learning algorithm, the input shape must be fixed. Therefore, to balance performance with computation overhead, choose only a subset of frames which give the most information regarding the video.

There are **three main approaches** to do such frame selection:

1. **uniform sampling** (fixed time intervals)
2. **scene change detection techniques**
3. **ordinary frame extraction** (keep only the specified number of frames)

Leveraging the `scenedetect API` (Python library which works directly with video clips), I have managed to implement some detectors which outputs the scenes where they detected a change in the scene. The available detectors:

- **ContentDetector** find ”fast cuts” or ”jump cuts” where there's an immediate and significant change in the visual content between consecutive frames using the weighted average of pixel changes in the HSV color space.
- **ThresholdDetector** used for identifying ”slow transitions” like fade-ins or fade-outs using the average pixel intensity in the RGB color space for each frame.
- **AdaptiveDetector** focuses on differences in content between adjacent frames, but it uses a rolling average of these differences instead of a fixed threshold.
- **HistogramDetector** finds fast cuts by comparing the histograms of consecutive frames, converting frames to the YUV color space and then focuses on the Y channel (luminance) to avoid color shifts.
- **HashDetector** utilizes perceptual hashing (compact numerical representation of an image that captures its visual characteristics) to determine the similarity between adjacent frames.

Choosing the right detector was more about trial-and-error. I have experimented with different video files to visualize how it selected the frames, and I have observed the fact that **there are some video clips which do not have significant changes between frames**, and neither one of the detectors have successfully extracted significant frames. This might be a problem if there is a significant amount of videos in this situation. Therefore, I chose the **AdaptiveDetector** due to its focus on adjacent frames.

Now the only thing left to do after choosing the selection method was to actually parse the directory with all the video clips, and select the **key frames**. I have set a parameter for the number of required frames in order to prepare the input for the next stage. Basically, I set the detector, for each one of the videos, I detected the key scenes, then if the number of key scenes did not match the one of the required frames, I have selected in-between frames to pad until the number of required frames was reached.

All the selected frames were carefully saved under directories with the same name as the video. I have also created a csv file containing metadata regarding the video clips and their frames. The csv file, `frames_metadata.csv` has a VideoID column to uniquely identify the video,and regarding the frames, `Key_Frames`, and `Total_Frames` columns which represent the number of key-frames and the total number of frames. There were no failed frame extraction and selection.

4.3 Feature Extraction using the Selected Frames

First of all, I had to make sure that all the frames have the same size, so I performed a resizing of 224 pixels by 224 pixels which is actually the common input size for pre-trained CNN model, VGG16. I have also converted the PIL Image Object to a 3D NumPy array (height, width, channels), and finally called an important method `preprocess_input` which handles this NumPy array properly for the VGG16 model (imported from `tensorflow.keras.applications.vgg16`).

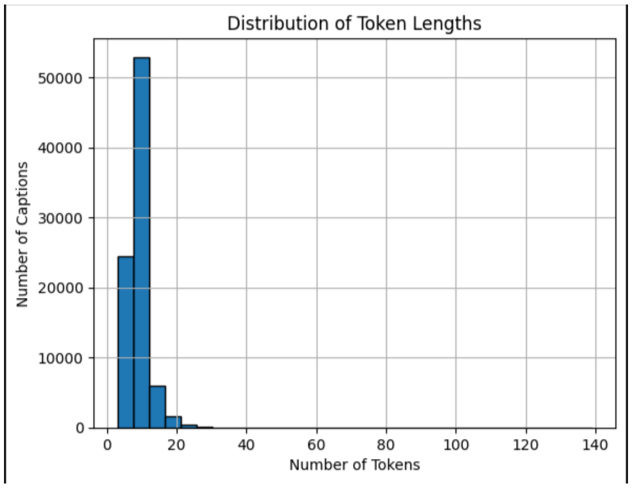


Figure 2: Distribution of Token Lengths

After preparing the input, I have loaded the pre-trained VGG16 model on the **ImageNet dataset**. I was not interested in its classifications, but in its capacity of extracting features, so I have accessed the second-to-last layer in the model which is usually another Dense layer (often with 4096 units) that acts as a high-level feature representation before the final classification. So the "result" is actually a 4096-dimensional vector for VGG16's for the (224x224x3) "image". Then for all of the frames within a video, I have applied this feature extractor and I appended all the resulting features to a NumPy file, saved again under its VideoID.

4.4 Captions Processing

This stage focused on performing various steps of cleaning applied on the annotations, and concatenating some special characters to the sentence (the annotations are sentence-level descriptions): **<BOS>** and **<EOS>** (mentioned earlier in 2.3).

After having the pre-processed data, I focused on transforming it into tokens using the **Keras Tokenizer** (word-level). Basically, it splits text into individual words based on whitespace and punctuation, and builds a vocabulary by assigning a unique integer to each unique word in the training corpus. When it encounters an "out-of-vocabulary" (OOV) word (a word not seen during its fit_on_texts step), it replaces it with a single oov_token (e.g., <unk>) if specified, or simply skips it. For my application, I have initialized my tokenizer to allow only top 1500 frequent words learned during the fit_on_texts step, and any other word to be considered unknown.

After the descriptions of the video were transformed to tokens, I have performed an analysis on the length of these tokens, which can be observed in 1 and 2.

Table 1: Token Length Statistics

Statistic	**Value**
Max Token Length	139
Min Token Length	3
Mean Token Length	9.10
90th Percentile	12
95th Percentile	14

Based on this analysis, I have decided to filter the data based on both a maximum and a minimum length to discourage the presence of outliers. To ensure a fixed size input, like I did with the number of frames, I have padded the sequences with white-spaces to reach the maximum length. An example of such sequence can be seen in 2.

Table 2: Video Description and Padded Sequence Examples

Description	**PaddedSequence**
⟨bos⟩ a bird is bathing in a sink ⟨eos⟩	[3, 2, 253, 5, 554, 9, 2, 465, 4, 0, 0, 0, 0, 0]
⟨bos⟩ a bird is splashing around under a running faucet ⟨eos⟩	[3, 2, 253, 5, 1, 81, 318, 2, 47, 903, 4, 0, 0, 0, 0]
⟨bos⟩ a bird is bathing in a sink ⟨eos⟩	[3, 2, 253, 5, 554, 9, 2, 465, 4, 0, 0, 0, 0, 0]

4.5 Caption Generator

Once both the sequences (captions) and frame features are ready, the final stage of the project can begin.

4.5.1 Model Architecture: LSTM Encoder-Decoder

The **LSTM Encoder** is responsible for processing the input sequence (in this case, video features) and compressing the information into a fixed-size "context vector" (represented by the final hidden and cell states). This context vector acts as a summary of the entire input sequence. The important aspects regarding this part of the architecture are:

- **LSTM layer**
- **input size** → dimensionality of each individual feature in the input sequence
- hidden size → number of hidden units in the LSTM layer
- outputs → output of the LSTM at each time step
- hidden → final hidden state of the LSTM (compact representation of the entire input sequence)
- cell → final cell state of the LSTM, which is also part of the internal memory of the LSTM

The LSTM Decoder takes the context vector from the encoder and generates an output sequence (in this case, a textual caption). It works by predicting one word at a time, using the previously predicted word and the encoder's context. The important aspects regarding this part of the architecture are:

- embedding size → dimensionality of the word embeddings
- hidden size → number of hidden units in the decoder's LSTM layer, which should typically match the encoder's hidden size
- vocabulary size → total number of unique words in the vocabulary (including special tokens)
- outputs, hidden, cell → predicted word logits for the sequence, along with the final hidden and cell states

The final model, **VideoCaptioningModel** which comprises both the encoder and decoder. The video features are first passed through the encoder to obtain the initial hidden and cell states. These initial states are then passed to the decoder along with the the caption tokens. The decoder generates the predicted word logits. The final output of the model is the sequence of predicted word logits, which can then be used to calculate loss during training or determine the most likely words during inference.

4.5.2 VideoCaption Dataset

I have created also a custom Torch Dataset for my problem, and a Dataloader. The Dataset takes into account important fields, like the metadata containing the unique identifier of the video, and the padded sequence associated to it, along with the extracted video features, and the optional tokenizer. Moreover, the Dataloader calls a collation function to combine individual samples into a single batch. This is particularly important when samples have varying lengths, as deep learning models typically require fixed-size inputs within a batch.

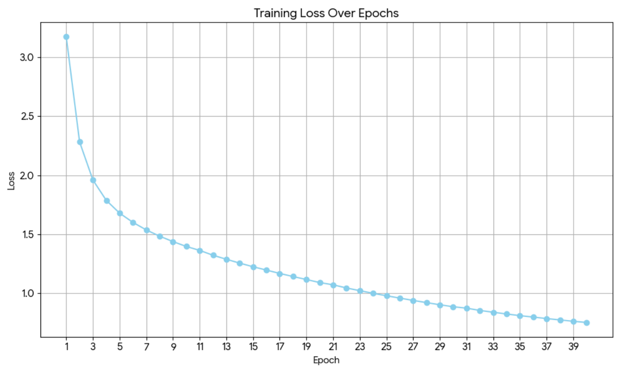


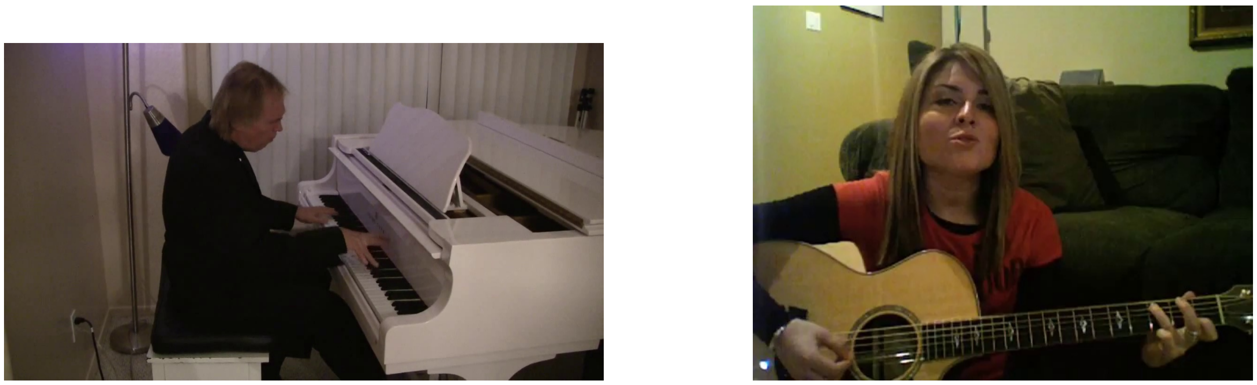
Figure 3: Training Loss over Epochs

4.5.3 Training

The model was trained for **40 epochs**, but an instance of the model was saved at epoch 10, 20, 30 and 40 to interpret the results, and observe the performances. The loss of the training can be observed in the 3. It is visible that it decreases over the epochs. The consistent decrease in loss indicates that the model is successfully learning from the training data.

4.5.4 Evaluation

Finally, I have implemented a method for the inference process to visualize the results, along with the actual video, and the generated caption. Unlike training, where the model is fed the correct caption sequence, during inference, the model must generate the caption word by word. An example of the inference process can be observed in the 4b and 4a. After visualizing some results, I have computed the **BLEU score**(BiLingual



(a) ”a man is playing a piano” (b) ”a young man is playing the guitar”

Figure 4: Examples of generated captions for video frames.

Evaluation Understudy) which is widely known for evaluating the quality of text generated by a machine against a set of human-created reference texts. A higher BLEU score generally indicates better quality. The model trained for only 10 epochs seem to have the highest BLEU score among all the other ones. Still there is room for improvement.

Table 3: Average BLEU Scores on Test Set

Model trained for	**Average BLEU Score**
10 epochs	0.1734
20 epochs	0.1685
30 epochs	0.1586
40 epochs	0.1548

4.6 Alternatives: BERT Tokenizer

References

[1] S. Amirian, K. Rasheed, T. R. Taha, and H. R. Arabnia, "Automatic image and video caption generation with deep learning: A concise review and algorithmic overlap," *IEEE Access*, vol. 8, pp. 218386–218400, 2020.

[2] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2625–2634, 2015.

[3] vtrnanh, "MSVD Dataset Corpus." https://www.kaggle.com/datasets/vtrnanh/msvd-dataset-corpus/data?select=video$_{c}orpus.csv$, 2023.

[4] S. Jain, "MSVD Clips." https://www.kaggle.com/datasets/sarthakjain004/msvd-clips, 2021.