

Semantic Video Frame Search Using OpenAI CLIP and Vector Database

Learn how to implement semantic video frame search using OpenAI CLIP and vector databases to efficiently find and retrieve specific visual content from videos.

Contents

01	Introduction	3
02	Set Up the Server	3
03	Get All Representative Frames from Video Scenes	4
04	Semantic Video Frame Search using CLIP	6
05	Speed Up Searching Experience with Vector Database	10
06	Similar Code Use Cases	13
07	Conclusion	14

Introduction

Semantic video frame search is an emerging generative task that locates and extracts video frames from a collection of files. Each frame search uses the presence of specific keywords, objects, themes or sentiments to generate results. As a result, you can use Semantic video frame search in data-related tasks such as social media content auditing, brand insights, and policy enforcement on user-generated videos.

This article explains how to perform semantic video frame search using CLIP on a Vultr Cloud GPU server. You will perform scene detection tasks, image/text embedding using the CLIP model, and apply vector databases to modify the search experiences with better industrial standard results.

Prerequisites

Before you begin:

- Deploy a [Ubuntu 22.04 A100 Vultr Cloud GPU server](#).
- Use [SSH to access the server](#) as a [non-root user with sudo privileges](#)
- Install JupyterLab.

Set Up the Server

1. Install all required dependency packages using Pip. These include

`scenedetect`, `opencv-python`, and `matplotlib`.

CONSOLE

```
$ pip install scenedetect torch opencv-python transformers chromadb ipywidgets
```

The above command installs the following semantic video search dependency packages:

- `scenedetect`: Scans for scene changes and cuts by analyzing a video input.
- `opencv-python`: Provides a wide range of image processing functionalities.
- `transformers`: Provides APIs and pipelines to download and use pre-trained transformer-based models.
- `chromadb`: A vector-based database for building AI applications.
- `ipywidgets`: Provides interactive HTML widgets for Jupyter Notebook using the IPython kernel.

2. Download a sample video to use as the input file. Replace the Google Chromecast video link with your desired file URL and save it as `video.mp4`

CONSOLE

```
$ wget http://commondatastorage.googleapis.com/gtv-videos-bucket/sample/ForBiggerFun.mp4 -O video.mp4
```

3. Copy the video file to the Jupyter user home directory.

CONSOLE

```
$ sudo cp video.mp4 /home/jupyter/
```

Get All Representative Frames from Video Scenes

1. Select Notebook and create a new `Python3` Kernel file.
2. Import all required model libraries in a new Notebook code cell.

PYTHON

```
from scenedetect import detect, AdaptiveDetector
from PIL import Image
from transformers import CLIPProcessor, CLIPModel
import chromadb
import cv2
import numpy as np
```

3. Press Shift + Enter to run the code cell and import all libraries.
4. Create a new function to fetch the representative frame from a video scene.

PYTHON

```
def get_single_frame_from_scene(scene, video_capture):
    frame_id = (scene[1] - scene[0]).frame_num // 2 +
    scene[0].frame_num
    video_capture.set(cv2.CAP_PROP_POS_FRAMES, frame_id)
    _, frame = video_capture.read()
    return Image.fromarray(cv2.cvtColor(frame,
    cv2.COLOR_BGR2RGB))
```

5. Create another function to extract all scenes using a `for` loop and the corresponding keyframes from an input video using the `PySceneDetect` module.

PYTHON

```
def get_frames_from_video(video_path):
    res = []
    video_capture = cv2.VideoCapture(video_path)
    content_list = detect(video_path, AdaptiveDetector())
    for scene in content_list:
        res.append(get_single_frame_from_scene(scene,
        video_capture))
    return res
```

6. Import your sample video as the input to fetch all representative video frames and store them in memory using a `frames` list that acts as the search space.

PYTHON

```
local_video_path = "/hone/jupyter/video.mp4"  
frames = get_frames_from_video(local_video_path)
```

In this section, you have loaded the sample video, detected available video scenes, and extracted the keyframes from each video scene for representation. Visit the [Video Scene Transition Detection and Split Video Using PySceneDetect](#) guide for more information on how to detect scenes. In the additional sections, modify the represented data using the CLIP model and a vector database.

Semantic Video Frame Search using CLIP

In this section, build a basic semantic video frame search system using CLIP. The model inputs a plain text query and returns the most suitable video frame based on the query's semantic meaning.

1. Load the pre-trained OpenAI CLIP model and processor from the transformers package.

PYTHON

```
model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")  
clip_processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
```

Within the above command:

- `CLIPModel`: Wraps up a vision model to generate visual features and a language model to generate text features.
- `CLIPProcessor`: Wraps up the vision and language models including the `CLIPFeatureExtractor` to encode images for the vision model, and the `CLIPTokenizer` to encode text for the language model.
- The sample code loads the Vision Transformer (ViT) base version with a `32` patch size using the `ViT-B/32` Transformer architecture as an

image encoder and a masked self-attention Transformer as a text encoder. Visit [the Hugging Face CLIP model card](#) for more available pre-trained models you can use.

Ignore any model-loading `TensorRT` warning messages.

2. Define a query prompt in plain text to use with the frame search. For example, `a walking penguin`.

```
PYTHON
```

```
query_text = "a walking penguin"
```

3. Pass the query text and all frames into the CLIP process as the model inputs.

```
PYTHON
```

```
inputs = clip_processor(text=query_text, images=frames,  
return_tensors="pt")
```

4. Pass the processed input to the CLIP model.

```
PYTHON
```

```
outputs = model(**inputs)
```

5. Get the raw output value, logits, as the unnormalized similarity score between the semantic meaning of the query text, and the visual content from each frame.

```
PYTHON
```

```
logits_per_image = outputs.logits_per_image
```

6. Normalize the logits among all frame-to-query pairs using the softmax function

```
PYTHON
```

```
probs = logits_per_image.softmax(dim=0)
```

7. Get the frame index with the highest normalized similarity score.

```
PYTHON
```

```
image_idx = np.argmax(probs.detach().numpy())
```

8. Check the searched image result.

```
PYTHON
```

```
frames[image_idx]
```

Verify that your output frame includes a cute walking penguin inside a TV screen cut from the input video.



9. Bind all frames together to build a function for a one-line semantic frame search.

PYTHON

```
def search_frame(query, all_frames):  
    inputs = clip_processor(text=query, images=all_frames,  
return_tensors="pt")  
    outputs = model(**inputs)  
    logits_per_image = outputs.logits_per_image  
    probs = logits_per_image.softmax(dim=0)  
    image_idx = np.argmax(probs.detach().numpy())  
    return all_frames[image_idx]
```

10. Conduct the semantic frame search using your function in a single line, but a different query prompt such as `jumping on the bed`.

PYTHON

```
search_frame("jumping on the bed", frames)
```

Verify another frame with a girl jumping on the bed displays in your output based on the new query prompt.



Speed Up Searching Experience with Vector Database

Speed up the video frame search process with a vector database to verify the performance limitations of the base semantic frame search function. Integrate a vector database into the system to bring its performance to the industrial standards as described in the steps below.

1. View the required running time for returning the frame result.

```
PYTHON
```

```
%%time  
search_frame("jumping on the bed", frames)
```

Output:

```
CPU times: user 5.31 s, sys: 765 ms, total: 6.07 s  
Wall time: 6.38 s
```

Based on the above output, the running time is `6.38 seconds` where the exact number may vary between `5 seconds` and `8 seconds` for different rounds. This is a long duration for a large-scale semantic image search system that supports millions of video frames in the search space with hundreds of query requests in a single session.

A practical industrial system generally requires a running time of less than `100 ms` for every single request. As a result, a vector database solves related time challenges by storing and indexing representations in the search space.

2. Load the ChromaDB vector database client package.

```
PYTHON
```

```
client = chromadb.Client()
```

3. Create a new ChromaDB collection to store your frame embeddings and corresponding IDs.

PYTHON

```
collection =  
client.get_or_create_collection("video_frame_embeddings")
```

4. Define a new function to get image embeddings using the CLIP model.

PYTHON

```
def get_image_embedding(image):  
    inputs = clip_processor(images=image,  
return_tensors="pt")  
    image_embeddings = model.get_image_features(**inputs)  
    return  
list(image_embeddings[0].detach().numpy().astype(float))
```

5. Get the image embeddings and IDs for all the extracted video frames.

PYTHON

```
image_embeddings, image_ids = [], []  
for i, frame in enumerate(frames):  
    embedding = get_image_embedding(frame)  
    image_embeddings.append(embedding)  
    image_ids.append(str(i))
```

6. Store the image embeddings and IDs in the ChromaDB collection.

PYTHON

```
collection.add(embeddings=image_embeddings, ids=image_ids)
```

7. Define a new function to get the query text embeddings in a similar way using the CLIP model and the CLIP processor.

PYTHON

```
def get_text_embedding(query):
    inputs = clip_processor(text=query, return_tensors="pt")
    text_embeddings = model.get_text_features(**inputs)
    return
list(text_embeddings[0].detach().numpy().astype(float))
```

8. Query ChromaDB to get the semantic frame search result.

PYTHON

```
%%time
query_text = "jumping on the bed"
query_embedding = get_text_embedding(query_text)

query_result =
collection.query(query_embeddings=query_embedding,
n_results=10)
frame_index = int(query_result["ids"][0][0])
frames[frame_index]
```

Verify that the same frame with a girl jumping on the bed displays in your session based on a similar `jumping on the bed` prompt. However, with a different running time as compared to the first result.

```
CPU times: user 24.9 ms, sys: 3.67 ms, total: 28.6 ms
```

```
Wall time: 31.2 ms
```



Based on the new result time of `31.2ms`, the frame search process improves by `100x` times as the running time meets the recommended industrial standards. However, due to the randomness of the ANN (Approximate Nearest Neighbors) algorithm, the ChromaDB result may not be consistent.

Similar Code Use Cases

In addition to the semantic video frame search, the code implementation in this article is also useful in other similar use cases such as the following.

- `Objectionable video frame detection`: Verifies if newly uploaded videos contain objectionable frames that are not suitable for some audience groups. For example, adult content is not allowed for infant audiences. You can create policy descriptions or keywords as the text query and use the search pipeline to verify the distance to the most similar video frames.
- `Keyframe searching for embedded advertisements`: Locates the frames where items appearing in a video can be recommended to audiences for online purchases or advertising. For example, a specific dress in a video could be a top-selling item. As a result, you can identify the specific product keyframes in the video using the semantic frame searching system.

Conclusion

You have set up a semantic video frame search system using the OpenAI CLIP model and a vector database on a Vultr Cloud GPU server. You identified target video frames and performed scene extraction. Then, used the CLIP model and processor to build a basic frame search system and test the performance limitation as compared to a vector database. Download the [Jupyter Notebook file](#) to verify the model operations in this article.



VULTR

