

Creating a semantic video search with OpenAI's Clip



Antti Havanko · [Follow](#)

6 min read · Feb 26, 2023



42

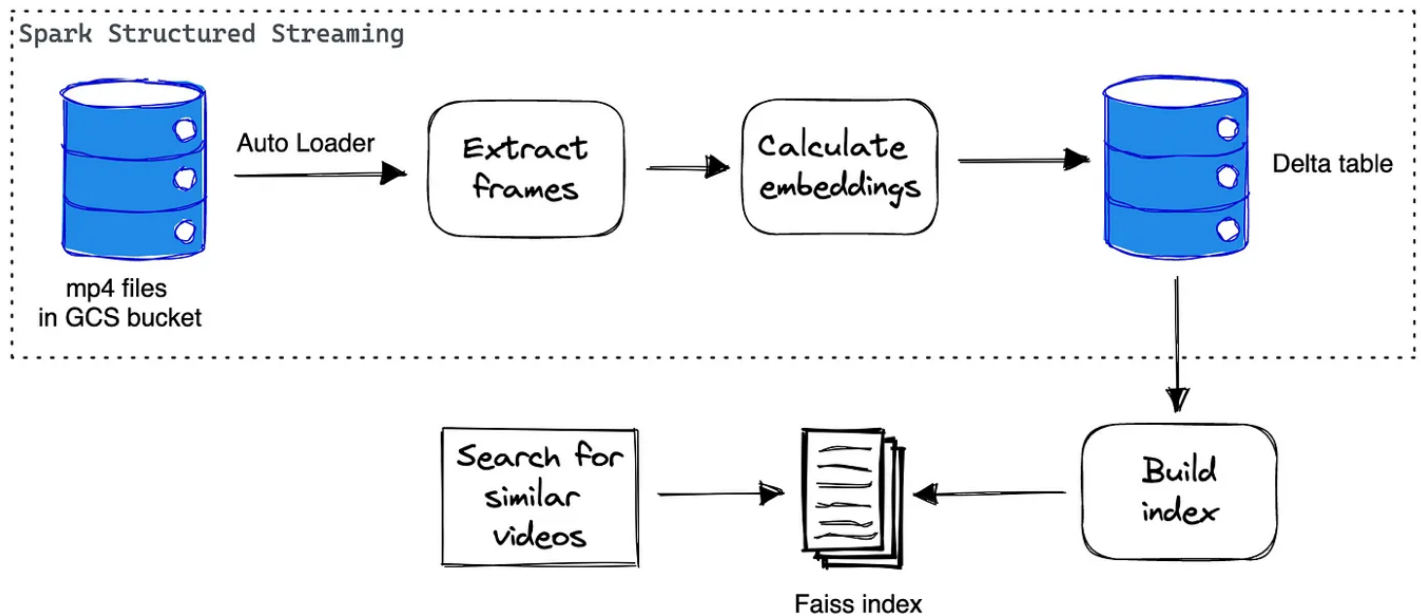


2



In today's world, video data is everywhere, and it contains valuable insights that can be used for a wide range of applications. Extracting these insights from the video data can be a daunting task, but with the help of cutting-edge technologies like the OpenAI's Clip and Spark, it can be made more manageable.

In this blog post, we will show you how to use the Clip model to generate embeddings to understand the videos, and then use Spark to create a streaming job to put everything together. Finally, we will build a vector search to find the most relevant frames using text.



By extracting frames from videos and understanding their content using embeddings, we can efficiently search for relevant video content using natural language queries. This system has many potential applications, such as media monitoring, video content analysis, and search engines.

Let's get started!

Extracting frames

The first step is to extract frames from the videos and store them as images. Each frame can be considered as a still image, and by processing all frames in the video, we can analyze the content of the entire video. However, generating embeddings for all the frames is a resource-intensive task and therefore, we will take a subset of frames from each video. We will be using only 10 random frames per video, but you can increase this number to improve the results.

To extract the frames from the videos, we will create a User-Defined Function (UDF), which will receive the video as binary data and extract frames using ffmpeg. Ffmpeg is a powerful tool for processing video and can extract frames at a high speed.

```
import ffmpeg
import io
import numpy as np
from PIL import Image
from pyspark.sql.types import ArrayType, BinaryType
```



 Search

 Write

Sign up

Sign in



```
w_percent = (target_width / float(width))
h_size = int((float(height) * float(w_percent)))
return target_width, h_size
```

```
@udf(returnType=ArrayType(BinaryType()))
def get_frames(content):
    with tempfile.NamedTemporaryFile() as f:
        f.write(io.BytesIO(content).getbuffer())

    probe = ffmpeg.probe(f.name, threads=1)
    video_info = next(s for s in probe['streams'] if s['codec_type'] == 'video')
    width, height = get_scaled_size(int(video_info['width']), int(video_info['height']))

    out, err = (
        ffmpeg
        .input(f.name, threads=1)
        .filter('scale', width, height)
        .output('pipe:', format='rawvideo', pix_fmt='rgb24')
        .run(capture_stdout=True, quiet=True)
    )
    frames = (
        np
        .frombuffer(out, np.uint8)
        .reshape([-1, height, width, 3])
    )
```

```
indexes = np.random.randint(frames.shape[0], size=10)
return [ to_byte_array(frame) for frame in frames[indexes, :] ]
```

After extracting the frames, we will sample 10 random frames and resize them to 224px wide because this is the input size of the Clip model, which we will use later in the process.

Generating embeddings

Next we will focus on using the Clip model from OpenAI to create embeddings for the images. These embeddings will be used to find the most similar frames for a given text or image search query.

The Clip model is a neural network that was trained on a large number of image and text pairs. Due to its training, the model has learned the “connection” between images and their associated text. It can embed images and text into a joint semantic space, which allows us to use it for finding the most similar image for a given text or image. This joint semantic space is what enables the Clip model to be used for various image and text-related tasks.

To create embeddings for the extracted images, we will be using the sentence-transformers library, which provides a simple interface for loading and interacting with the Clip model.

We create another UDF that encodes the images using Clip by taking in the

images and returning an embedding for each image:

```
import io
from PIL import Image
from pyspark.sql.types import ArrayType, FloatType
from sentence_transformers import models, SentenceTransformer

img_model = SentenceTransformer(modules=[models.CLIPModel()])

@udf(returnType=ArrayType(ArrayType(FloatType())))
def get_embeddings(frames):
    images = [ Image.open(io.BytesIO(frame)) for frame in frames ]
    vectors = img_model.encode(images)
    return [ vector.tolist() for vector in vectors ]
```

Once we have created embeddings for each image, we can store them in a Delta table. This enables us to easily query the embeddings when building the similarity search by using Spark's SQL API for querying.

Creating ETL on Databricks

In the previous chapters, we have covered the steps required to extract frames from the videos, create embeddings using the Clip model and store the results in a Delta Table. Now we will use Spark Structured Streaming to put everything together.

The first step is to listen for incoming files using Databricks Auto Loader, which is a managed service for ingesting data from cloud storage. Auto Loader can process files as they arrive in a cloud storage bucket, and it

keeps track of the ingestion progress, so each file is processed only once. To use Auto Loader, we specify the cloud storage path and file format, and then we can start reading the incoming files using Spark's `readStream` method. In our case, our videos are stored as MP4 files, so we specify the path filter accordingly:

```
(
  spark
    .readStream.format("cloudFiles")
    .option("cloudFiles.format", "binaryFile")
    .option("pathGlobFilter", "*.mp4")
    .load("gs://...")
)
```

Once we have the incoming files, we need to extract frames and embeddings for each video using the UDFs we defined earlier. We can call these UDFs as columns in a Spark DataFrame, and Spark will automatically apply them to each row:

```
(
  spark
    .readStream.format("cloudFiles")
    .option("cloudFiles.format", "binaryFile")
    .option("pathGlobFilter", "*.mp4")
    .load("gs://...")
    .withColumn('frames', get_frames(col('content')))
    .withColumn('embeddings', get_embeddings(col('frames')))
    .select('path', 'frames', 'embeddings')
)
```

In this example, we add two new columns to the DataFrame: “frames” and “embeddings”. The `get_frames` function extracts frames from the video content, and the `get_embeddings` function creates embeddings for the frames using the Clip model. We also select the path column, which contains the URI of the video file in the cloud storage bucket.

Finally, we write the resulting DataFrame to a Delta Table using Spark’s `writeStream` method. We specify the output format and mode, as well as the checkpoint location for fault tolerance.

```
(  
    spark  
        .readStream.format("cloudFiles")  
        .option("cloudFiles.format", "binaryFile")  
        .option("pathGlobFilter", "*.mp4")  
        .load("gs://...")  
        .withColumn('frames', get_frames(col('content')))  
        .withColumn('embeddings', get_embeddings(col('frames')))  
        .select('path', 'frames', 'embeddings')  
        .writeStream  
        .format("delta")  
        .outputMode("append")  
        .option("checkpointLocation", "gs://...")  
        .start("gs://...")  
)
```

The data is then stored in a Delta Table, which is located in our Google Cloud Storage bucket. The system now processes incoming videos in near-

real-time and stores the frames and embeddings for use. There are three columns in the table: 1) GCS URI, 2) extracted frames (images), 3) embeddings from Clip model:

Building search index

After processing and storing the video frames and their embeddings to a Delta Table, the next step is to build an index for efficient similarity search. For this task, we are using the txtai library, which provides a simple interface for indexing and searching embeddings using Faiss as the backend.

To build the index, we need to load the embeddings from our Delta Table and convert them into the format expected by txtai. We create a data generator function that yields each embedding as a numpy array with a unique ID. Once we have the generator function, we can pass it to the Embeddings class from txtai. In this case, we use the “external” method since we are providing the embeddings ourselves. Txtai can also generate

the embeddings itself using various pretrained models.

```
import numpy as np
from txtai.embeddings import Embeddings

def data_generator(iter):
    for row in iter:
        path = row['path']

        for idx, e in enumerate(row['embeddings']):
            index_id = f"{path}|||{idx}"
            yield index_id, np.array(e), None

embeddings = Embeddings({"method": "external", "transform": transform, "content":
df_iter = sql("SELECT path, embeddings FROM videos.embeddings").rdd.toLocalIterator
embeddings.index(data_generator(df_iter))
```

The final step is to perform a similarity search using the query text. We encode the query using the Clip model as well to the embedding for our search query. We group the results by the video URI and sort them by the total similarity score for each video to get the most relevant videos.

```
def group_by_video(results):
    data = {}
    for index, prob in results:
        video_uri = index.split("|||")[0]

        if video_uri not in data:
            data[video_uri] = []
        data[video_uri].append(prob)
```

```
# sort by total probability
return list(sorted(data, key=lambda k: sum(data[k]), reverse=True))

query = "american football"
query_vector = text_model.encode([query])[0]
results = embeddings.search(query_vector, 30)
video_uris = group_by_video(results)

for video_uri in video_uris[:3]:
    display.display(display.Video(video_uri, width=300))
```

Overall, our pipeline allows us to process videos in near-real time, extract frames and embeddings, store them in a Delta Table, and perform efficient similarity search using a vector index. This opens up many possibilities for applications such as video search, recommendation systems, and more.

[Machine Learning](#)[Databricks](#)[Embedding](#)[Similarity Search](#)[Video Search](#)



Written by Antti Havanko

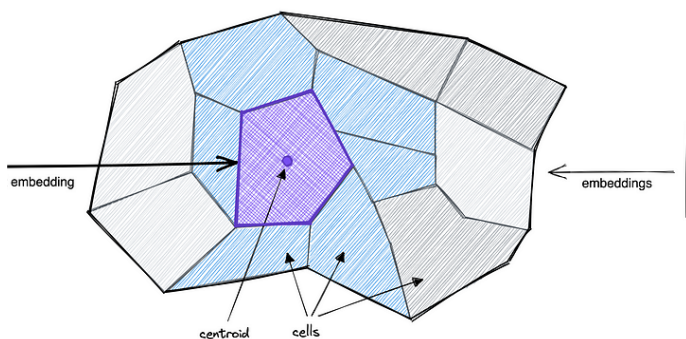
92 Followers

Manager. Maker of things. Optimist. Happy.

Follow



More from Antti Havanko



 Antti Havanko


Building Image search with OpenAI Clip

OpenAI's Clip is a neural network that was trained on a huge number of image and text...

5 min read · Apr 9, 2022

 125  1



 Antti Havanko

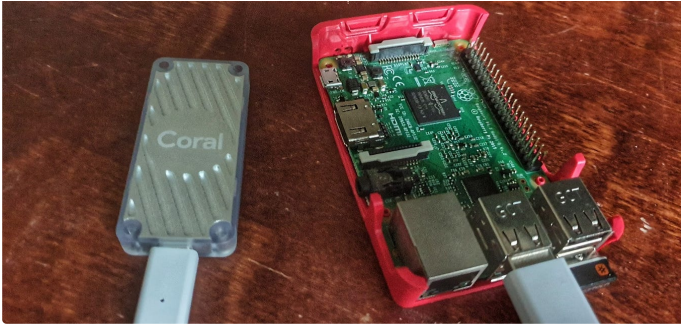
Time series forecasting for Prometheus & Grafana with...

Use BigQuery ML for adding forecasting capabilities to Prometheus and make your...


5 min read · Aug 14, 2021

 50  1





Goog Big Que

 Antti Havanko


Detect objects with Raspberry Pi using Google Coral Edge TPU

I finally managed to find time for testing the Coral USB Accelerator from Google. This...

3 min read · Sep 8, 2020

 24 



 Antti Havanko

Sentiment classification using BigQuery ML

Training machine learning model without a single line of code!

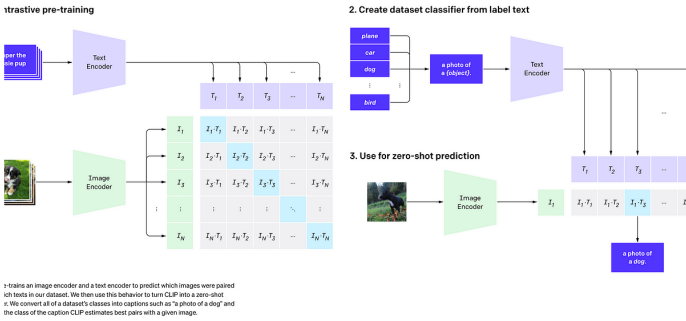
5 min read · Jan 16, 2021


 27 



See all from Antti Havanko

Recommended from Medium



 Szymon Palucha

Understanding OpenAI's CLIP model

CLIP was released by OpenAI in 2021 and has become one of the building blocks in many...

11 min read · Feb 24, 2024

 73  2



 Bright Money

Boosting Credit Scores through Smart Credit Utilization

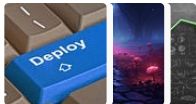
We will explore 5 psychological factors that influence positive credit behavior which lea...

7 min read · Apr 3, 2024

 52 



Lists



Predictive Modeling w/ Python

20 stories · 1318 saves



Natural Language Processing

1537 stories · 1071 saves



Practical Guides to Machine Learning

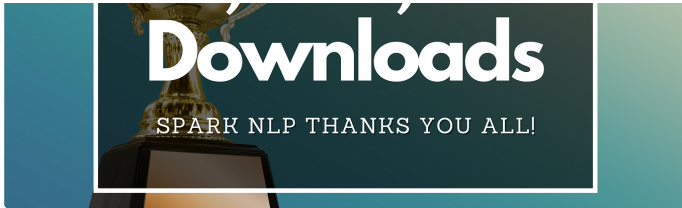
10 stories · 1585 saves



The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 413 saves





 Maziyar Panahi in spark-nlp

Spark NLP 5.2.0: Zero-Shot Image Classification by CLIP

Introducing a Zero-Shot Image Classification by CLIP, ONNX support for T5, Marian, and...

5 min read · Dec 28, 2023



9



 Federico Bianchi in Towards Data Science

Teaching CLIP Some Fashion

Training FashionCLIP, a domain-specific CLIP model for Fashion


10 min read · Mar 7, 2023



232

2



 Niranjana Akella

Combining ImageBind with Qdrant: Vector Similarity Search...

Build your very own Multimodal search engine using Imagebind and Qdrant vector...

7 min read · Feb 22, 2024



68

2



 Umair Ali Khan

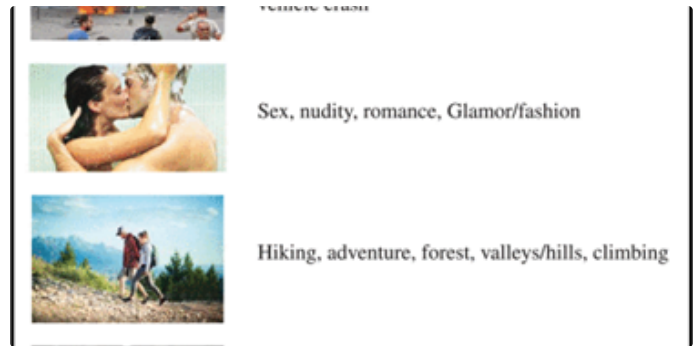
Exploring Joint Representations in Multimodal Learning

In 2017, as large-scale pre-trained models were gaining traction for image classificatio...

11 min read · Feb 2, 2024



25



See more recommendations

[Help](#) [Status](#) [About](#) [Careers](#) [Press](#) [Blog](#) [Privacy](#) [Terms](#) [Text to speech](#) [Teams](#)