Deep Scene

Author: R.J. Heijdens

Leveraging
Artificial Intelligence (A.I.)
for Scene Detection
in Broadcast Video.

JW Player
Advisers: K. Sindi & N. Yungster
Lecturer: E. Duipmans
Saxion University of Applied Sciences

June 2017

Abstract

Scene detection is the task to automatically segment an input video into meaningful and storytelling parts, without any help from the producer, using perceptual cues and multimedia features extracted from data. This thesis describes a framework, and its implementation, for an end-to-end deep learning based scene detection solution. Our model achieves results which are comparable to the state of the art on the task of scene detection by learning a distance measure between shots, which are series of frames that runs for an uninterrupted period of time. Notably our model achieved an average maximum intersection-over-union score of 0.48 on the BBC Planet Earth Planet Earth dataset. Furthermore we describe how our framework is implemented using TensorFlow.

Acknowledgements

I would first like to thank my colleagues at JW Player for providing- and supporting me with this graduation project. In particular I would like to single out my advisor Kamil Sindi for coming up with the idea of doing the project about scene detection and his crucial guidance during the project.

Furthermore I would like to thank Nir Yungster and Rick Okin for all the support I received during this project. I am grateful that they flew me to New York City in order to have me present this project to all the colleagues in the engineering department and in the Developer Lounge at the JW Insights conference.

Additionally, I would also like to thank my tutors Evert Duipmans and Jan Stroet, for their guidance during this project. Especially Evert's reviews of the thesis have been very helpful in making this thesis what it has become.

Contents

Abstract ii
Acknowledgements iii

1 Introduction **1**
1.1 Company Description 1
1.2 Problem Statement 2

2 Introduction to Deep Learning **4**
2.1 What is Machine Learning? 4
2.2 Neural networks 5
2.3 Deep Learning 7
2.4 Convolutional Neural Networks 8

3 Deeply-Learned Scene Detection **11**
3.1 Related Work 11
3.2 Architecture Overview 12
3.3 Feature Extraction 13
3.3.1 Shot detection 13
3.3.2 Visual features 15
3.3.3 Transcripts 15
3.3.4 Audio 16
3.3.5 Time 17
3.4 Loss Function 17
3.4.1 Constrastive Loss 19
3.4.2 Triplet Loss 20
3.4.3 Regularization 21
3.5 Clustering 22

4 Implementation **24**
4.1 TensorFlow 24
4.2 Microservice Architecture 25
4.3 API 25
4.4 Demo 26
4.5 Python Libraries 28
4.6 Deep Scene at scale 29

5 Results **30**
5.1 Performance metrics 30
5.2 Experiments setting 31
5.3 Evaluation 31

6 Development Process **33**
6.1 Version Control System 33
6.2 Bugtracking 33
6.3 Communication 33
6.4 Project agenda 33

7 Conclusion **35**
7.1 Future Work 35

8 Reflection **36**

A Task Overview **37**

Bibliography **40**

Chapter 1

Introduction

In this chapter we will introduce the reader to the problem statement of this thesis. In order to clarify the context of this study we begin with a short introduction of JW Player as a company. Next, we describe the problem publishers, the customers of JW Player, are facing when monetizing their media libraries.

1.1 Company Description

JW Player's mission is to provide video professionals with world-class technology to reach, grow and monetize their audiences.

JW Player's customer base mainly consists out of publishers, enterprises and (developer) agencies. Well known customers are PBS, The Washington Post, RedBull, Vice, Little Things, Univision, WWE, Hearst, Maker Studios and others.

Publishers standardize upon JW Player to deliver and monetize their content. Enterprises use JW Player to train their users and to market their products and developer agencies around the world integrate JW Player into their sites, apps and services.

JW Player's products can be subdivided into two branches: JW Player and JW Platform. The JW Player itself is a HTML5/JavaScript video player. The JW Player works in every popular browser, where it offers a premium user experience and complete developer control through APIs. Features of the JW Player include:

- Apple HTTP Live Streaming (HLS) and MPEG-DASH support.
- Advanced Advertisement scheduling.
- Digital Rights Management (DRM) support.
- Interactive Recommendations.

JW Platform on the other hand is a set of mass scalable cloud services for all video publishing components with an intuitive dashboard and complete developer API control. The following features can be found within the JW Platform:

- Adaptive video-on-demand (VOD) Streaming
- Data-Driven Video Recommendations
- Video Ad Insertion
- Live Streaming
- (Real-time) Video Analytics

Furthermore JW Player also offers multi-platform OTT (over-the-top) solutions in the form of JW Showcase. JW Showcase is a service that makes JW Platform content available everywhere. Currently OTT solutions include app builders for Apple TV, Web, Roku and iOS and Android.

1.2 Problem Statement

A lot of companies with (big) media libraries are trying to monetize their video content. Monetization of video content is often done by displaying one or more pre-roll ads in front of a video. While pre-roll ad creatives are a very effective way for building brand awareness, their impact on viewer engagement can be dramatic and often leads to viewers abandoning the video before the advertisement has been finished.

A better approach to monetizing video content could be to display advertisements after a certain period of time in a video. These so called mid-roll advertisements potentially enjoy a higher completion rate than both pre- and post-roll ad creatives. Since viewers have already watched a part of the video before being served an ad, it is more likely that they will remain engaged with the video until the advertisement has finished.

In order to minimize the effect of mid-roll advertisements on user engagement it is very important that they are being placed correctly in a video. In TV broadcasting it is common to place a mid-roll ad right after a cliffhanger, or in between **scene breaks**.

Scene breaks, also known as Logical Story Units (LSUs) [16], or simply story units [7], are a series of shots that communicate a unified action with a common locale and time [6]. Viewers perceive the meaning of a video at the level of story units [7], [28].

Scene detection is the task to automatically segment an input video into meaningful and story-telling parts, without any help from the producer, using perceptual cues and multimedia features extracted from data [38], [2].

The goal of this study is to implement a machine learning solution that can automatically segment a video into logical scene breaks. The locations of these scene breaks can then be used for automatic mid-roll advertising placement. Besides ad placement it should also be possible to use this solution for content indexing or the detection of closing credits in a video, but the focus in this study will be on leveraging the solution for advertisement insertion.

Our solution must be capable of learning feature representations [4] in order to prevent labour intensive manual feature engineering. The process of learning feature representations is better known as *Feature Learning* or *Deep Learning*.

To be concrete, we can derive the following main research questions from the preceding problem statement:

1. How can we measure semantic coherency between shots?
2. Given this semantic coherency measure, how can we cluster shots in such a way that we will end-up with meaningful and story-telling parts?

Chapter 2

Introduction to Deep Learning

In this chapter we will give the reader a brief introduction to *Deep Learning*. In order to explain the concept of *Deep Learning* we first need to introduce the reader to the broader concept of *Machine Learning*. After introducing Machine Learning we will introduce the reader to *Neural Networks*. Finally, we give a quick introduction to *Convolutional Neural Networks*.

2.1 What is Machine Learning?

Machine Learning is a method of data analysis that automates analytical model building. Using algorithms that iteratively learn from data, machine learning allows computers to find hidden insights without being explicitly programmed where to look. These insights allow machine learning algorithms to overcome following strictly static program instructions by making data-driven predictions or decisions.

Machine learning is employed in a range of computing tasks where designing and programming explicit algorithms with good performance is difficult or unfeasible. Well known applications of machine learning include: email spam filters, optical character recognition (OCR), computer vision, self-driving cars, fraud detection systems and content recommendation systems.

Machine learning tasks are typically classified into three broad categories, depending on the nature of the learning "signal" or "feedback" available to the learning system [30]:

- Supervised learning: The desired output of the machine learning algorithm for a given signal is available and can be used to deliver feedback to the learning system, which will allow the system to adapt. For example, when training a (Convolutional) Neural Network to predict the steering angle for a self-driving car based on an image of the road ahead and a known steering angle, which was recorded when taking the image, we let the network make a prediction and then penalize the correctness of that prediction based on the ground-truth data.

- Unsupervised learning: The desired output of the machine learning algorithm for a given signal is *not* available. Unsupervised learning algorithms can learn

to draw inferences by using supervised learning methods that is, it can learn to draw inferences based on data it has seen before.

- Reinforcement learning: Reinforcement learning algorithms learn the *condition-action* component [34]. Given a signal the algorithm outputs an action of which it will receive feedback (such as a hefty bill for rear-ending the car in front in case of a self-driving car) but is not told the correct action (to brake more gently and much earlier). Reinforcement learning algorithms will adapt based on the evaluation.

Another categorization of machine learning tasks arises when one considers the desired *output* of a machine-learning solution [5]:

- In *classification* the aim is to assign each of the inputs to one of a finite number of discrete categories.
- If the desired output consists of one or more continious variables, then the task is called *regression*.
- *Clustering* is the task of discovering and grouping similar inputs. Unlike in classification, the groups are not known beforehand, which makes this typically an unsupervised task.
- *Density estimation* finds the distribution of inputs in some space.
- *Dimensionality reduction* simplifies inputs by mapping them into a lower-dimensional space. Dimensionality reduction is one of the foundational pillars of Deep Scene as we will demonstrate in Chapter 3.

2.2 Neural networks

An artificial neural network is a computer system loosely modelled on the human brain and nervous system. The basic computational unit of the brain is a *neuron*. Approximately 86 billion neurons can be found in the human nervous system and they are connected with approximately 10^{14} - 10^{15} *synapses*. Figure 2.1 shows a cartoon drawing of a biological neuron and figure 2.2 a common mathematical model. Each neuron receives input signals from its *dendrites* and produces output signals along its (single) *axon*. The axon eventually branches out and connects via synapses to dendrites of other neurons. In the computational model of a neuron, the signals that travel along the axons (e.g. x_0) interact multiplicatively (e.g. $w_0 x_0$) with the dendrites of the other neuron based on the synaptic strength at that synapse (e.g. w_0). The idea is that the synaptic strengths (the weights w in the mathematical model) are learnable and control the strength of influence of one neuron on another. In the basic model, the dendrites carry the signal to the cell body where they all get summed. If the final sum is above a certain threshold, the neuron can *fire*, sending a spike along its axon. In the computational model, we assume that precise timings

of the spikes do not matter, and that only the frequency of the firing communicates information. Based on this *information rate* interpretation, we model the *firing rate* of the neuron with an **activation function** f, which represents the frequency of the spikes along the axon [1].

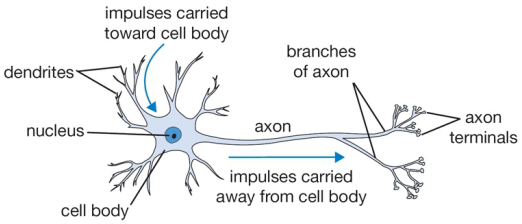

Figure 2.1: *Biological neuron.*

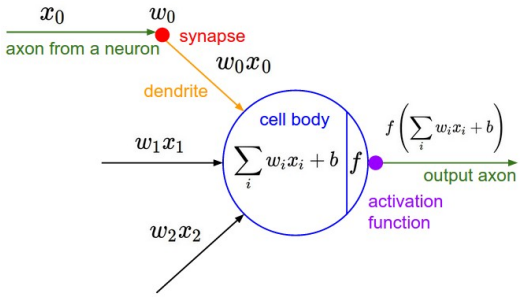

Figure 2.2: *Mathematical neuron.*

A commonly used activation function is the *Rectified Linear Unit*, better known as ReLU. The ReLU has become very popular in the last few years because it allows for faster and more effective learning on large datasets than the *sigmoid* [15] activation function, and other similar functions. It computes the function $f(x) = max(0, x)$. In other words, the activation is simply thresholded at zero, which makes it a very cheap activation function in terms of resources.

Neural networks are composed as a collection of neurons that are connected in an acyclic graph. Often neural networks are organized into distinct layers of neurons. For regular neural networks, the most common layer type is the *fully-connected layer* in which neurons between two adjacent layers are fully pairwise connected by weighted connections, but neurons within a single layer share no connections. In an artficial neural network all the neurons compute dot products between layers and thus the inputs and outputs of neural networks consists out of vectors. It is common to normalize the components of the input vector to the range of $[0, 1]$.

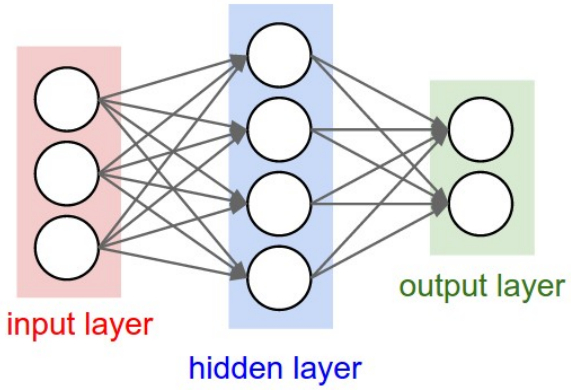

Figure 2.3: *Representation of a 2-layer Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs [19].*

[1] Introduction and images adapted from CS231n [19].

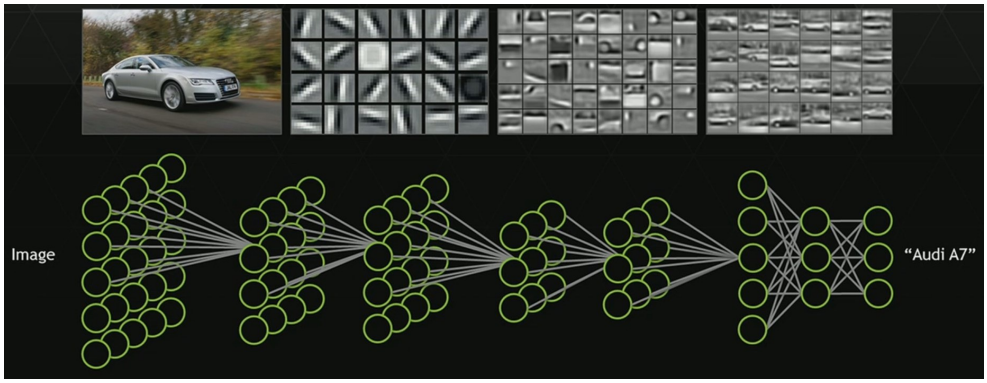

Figure 2.4: *Representation of a deep learning algorithm. We would like the raw input image to be transformed into gradually higher levels of representation, representing more and more abstract functions of the raw input, eg., edges, corners and contours, objects and parts, and finally a car. Image adapted from [23].*

Neural networks are trained using the *backward propagation of errors* or *back-propagation*. Back-propagation repeatedly adjusts the weights of the connections in a neural network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units [29].

2.3 Deep Learning

Formally deep learning algorithms are a class of machine learning algorithms that learn a distributed representation of its input data. In practice that means that deep learning methods aim to learning feature hierarchies with features from higher levels of the hierarchy formed by the composition of lower level features. For example, considering the lowest layer of the network to be the input layer, the lower layers of a *Convolutional Neural Network* learn to detect edges, the layers above that learn to detect corners and contours, and the layers above that objects and parts of objects. Automatically learning features at multiple levels of abstraction allows a system to learn complex functions mapping the input to the output directly from data, without depending completely on human-crafted features. This is especially important for higher-level abstractions, which humans often do not know how to specify explicitly in terms of raw sensory input [4].

In a simple case, there might be two sets of neurons: one set that receives an input signal and one that sends an output signal. When the input layer receives an input it passes on a modified version of the input to the next layer. In a deep network, there are many layers between the input and the output, allowing the algorithm to use multiple processing layers, composed of multiple linear and non-linear transformations. Deep

learning allows the computer to build complex concepts out of simpler concepts [13] [11].

There are huge numbers of variants of architectures for deep learning, but a common architecture is the *deep neural network* (DNN). A DNN is an artificial neural network (ANN) as described in 2.2 with multiple hidden layers of units between the input and output layers [4].

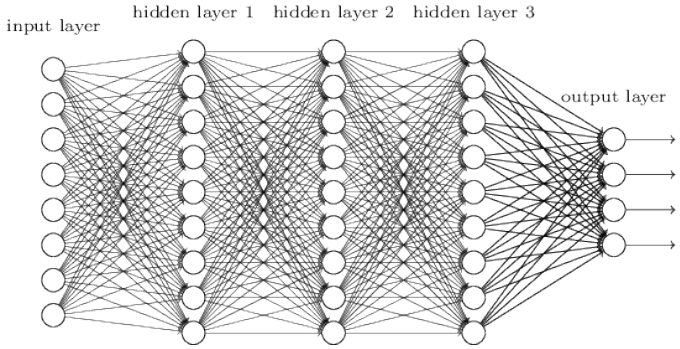

Figure 2.5: *A Deep Neural Network with 3 fully-connected hidden layers [26]. Note that every in the hidden layers has a connection to every other neuron in the preceding and proceeding layer.*

2.4 Convolutional Neural Networks

Convolutional Neural Networks (CNN's, or ConvNets) are special class of Neural Network architectures targeted at image processing. They are very similar to regular (deep) neural networks, but there are a couple of important differences. CNN's make the explicit assumption that their inputs consist out of images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network. In particular, unlike a regular Neural Network, the layers of a Convolutional Neural Network have neurons arranged in 3 dimensions: width, height, and depth.

At the core of a convolutional neural network are the convolutional layers. Convolutional layer's parameters consist out of a set of learnable filters (or kernels). Every filter is small spatially, but extends through the full depth of the input volume. During a forward pass we slide these filters (or more precisely, convolve) across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter we will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network [19]. Every layer consists out of multiple of these filters and will produce a separate activation map. We will stack these activation maps along the depth dimension and produce the output volume.

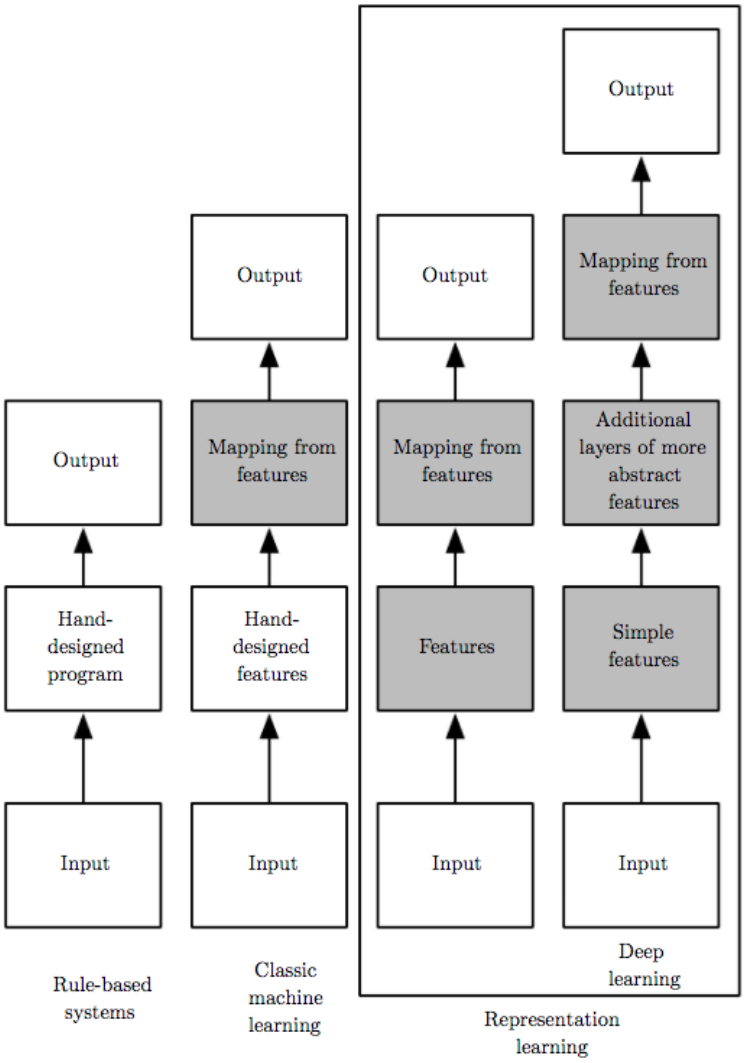

Figure 2.6: *Flowcharts showing how the different parts of an AI system relate to each other within different AI disciplines. Shaded boxes indicate components that are able to learn from data [13].*

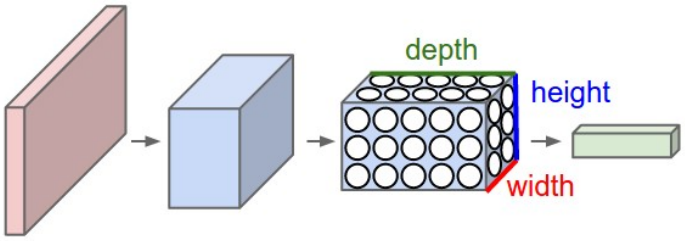

Figure 2.7: *A Convolutional Neural Network arranges its neurons in three dimensions (width, height, depth). Every layer of a CNN transforms the 3D input volume to a 3D output volume of neuron activations. In this example the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels). [19]*

For a more in-depth introduction to deep learning we would like to refer the reader to the excellent Deep Learning book by Goodfellow et al. [13] or to Learning Deep Architectures for AI by Y. Bengio [4].

Chapter 3

Deeply-Learned Scene Detection

In this chapter we will explain how deep learning can be used to detect scene breaks in video content. First we will describe related work, after which we will give a high-level overview of our architecture. Finally, we will perform a deep dive into the components of our architecture.

3.1 Related Work

Much research has been done into the segmentation of videos into scenes. Hanjalic *et al.* [16] defined the concept of a *logical story unit* and proposed a segmentation method that can provide a concise and comprehensive entrance level to an event-oriented movie-organization scheme.

In [33] a Scene Transition Graph (STG) is used. Shots are modelled as nodes in a graph, and every node has a connection to all other nodes. Then based on a (dis)similarity metric edges within the graph are removed until a given similarity threshold has been reached. At that point all the sub-graphs with only a single connection to another sub-graph can be considered as scenes.

Logical story units are subjective, to limit subjectivity Vendrig *et al.* [38] presented definitions for logical story units based on film theory. For evaluation of these definitions they introduced a method measuring the quality of a segmentation method and its economic impact, rather than the amount of errors.

Our work will build on top of the work of [2], in which a deep learning model is described that automatically partitions videos into semantically coherent scenes by learning a distance measure between shots by employing a Siamese neural network. Siamese neural networks, first introduced in 1994 in [8] by Bromley *et al.*, are networks that consist out of two identical sub-networks that share their parameters and weights. Siamese neural networks are capable of learning a distance measure within the embedding (output) space that approximates the neighbourhood relationships in the input space. They have proven to be effective for various use cases e.g. signature verification [8], image recognition [21] and person identification [37].

Furthermore we will also explore a second architecture based on a triplet network [18]. Triplet networks aim to learn useful representations by performing distance

comparisons between anchor, positive and negative exemplars. Where the anchor is a randomly picked exemplar, the positive exemplar is an exemplar belonging to the same category as the anchor and the negative is an exemplar belonging to a different category. The goal is then to minimize the distance between the anchor and positive exemplars, and to maximize the distance between anchor-positive and negative exemplars. In [18] it was proven that triplet networks are capable of learning a better representation than that of its immediate competitor, the Siamese network.

3.2 Architecture Overview

In this section we will give a high-level overview of the architecture of our framework for scene detection.

Brunelli [9] defines video structure as the decomposition of a stream into shots ("contiguously recorded sequences") and scenes. Given that a shot usually has uniform content, scene detection can also be viewed as the problem of grouping adjacent shots together, with the objective of maximizing the semantic coherence of the resulting segments. This poses scene detection as a clustering problem.

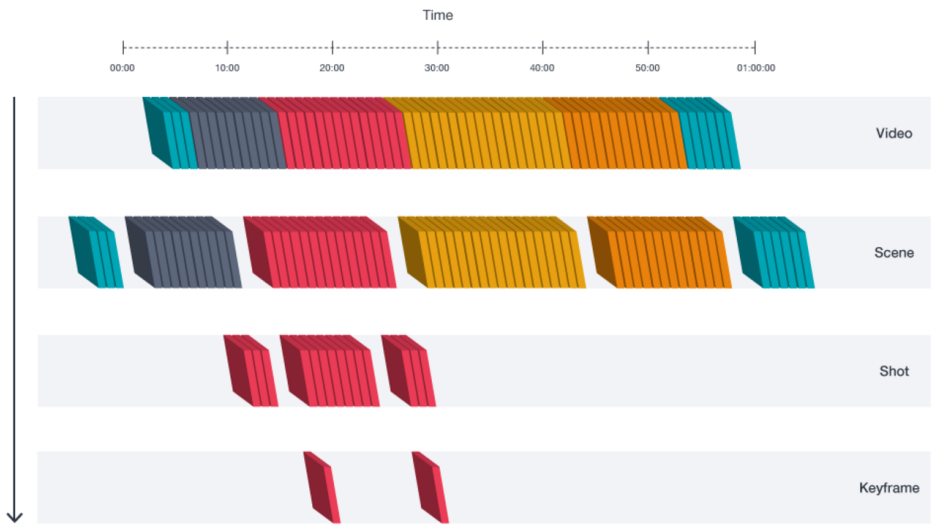

Figure 3.1: *Chart depicting the hierarchical structure of video. Videos can be decomposed in the following order: video* $\mapsto$ *scenes* $\mapsto$ *shots* $\mapsto$ *frames.*

Our model solves this problem by employing a two step approach: 1) we train a deep-learning model to learn a distance measure (i.e. similarity measure) between all pairs of shots by leveraging visual, audio and textual features extracted from the video; 2) we then cluster contiguous groups of shots into scenes based on the full similarity matrix of shots.

Before we can extract features from shots in the video, we first need to localize them in the video. Thus we start by segmenting a video into shots using an off-the-shelf shot detector, then for each of the shots we extract visual, audible, and textual

features. Next, we concatenate all the individual features and the normalized position of the shot in the video into a single feature vector. We then feed the feature vectors for adjacent shot pairs into a neural network whose task it is to learn a function that maps highly dimensional feature vectors to lower dimensional output vectors, which we call embeddings. The goal is then to condition this network in such a way that the euclidean distance between the embeddings of shots belonging to the same scene is smaller than shots belonging to different scenes. The rationale behind this is that if the distance between shots belonging to the same scene in the embedding space is smaller than the distance between shots belonging to different scenes we can use this distance (or similarity) measure to look for partitions of the video that maximize the similarity between contiguous segments of video. To learn this mapping function we employ a Siamese neural network [14].

Siamese neural networks consist out of two identical branches with exactly the same architecture that share their weights and have proven to be capable to learn such a function, we will further explain how Siamese networks are capable of learning such a mapping in Section 3.4.1.

During our research we found out that triplet networks, which are very similar to Siamese networks, are capable of learning better a better embedding function than Siamese networks, and thus later versions of the model use a triplet loss function rather than the contrastive loss function which is used in Siamese networks. We will further explain the working of triplet networks in Section 3.4.2.

The Siamese network (and in later versions the triplet network) then produces a embedding (low dimensional vector encoding) for each of the shots on which we can perform the clustering step. After collecting the embeddings for all the found shots in a given video we construct a similarity matrix of the shots by calculating the cosine similarity between all the shot embeddings. Finally, we cluster adjacent shots in the similarity matrix in order to produce the final scene boundaries as described in section 3.5. An overview of the architecture is depicted in Figure 3.2.

3.3 Feature Extraction

The task of scene detection requires good representations of the video content. In order to construct a good representation of shots extract both visual features, audible features and textual features from the shots in the video. We concatenate all the extracted features into a single feature vector per shot, which is then fed into our neural network that learns the distance measure between shots.

3.3.1 Shot detection

Before we can start extracting features we need to segment the video into shots. We do this by employing an off-the-shelf shot detector. Specifically, we utilize the ContentDetector in the PySceneDetect library, which despite its name does not detect scenes but finds shot boundaries. The ContentDetector detects fast cuts between shots by measuring changes in colour and intensity between frames, if a change exceeds

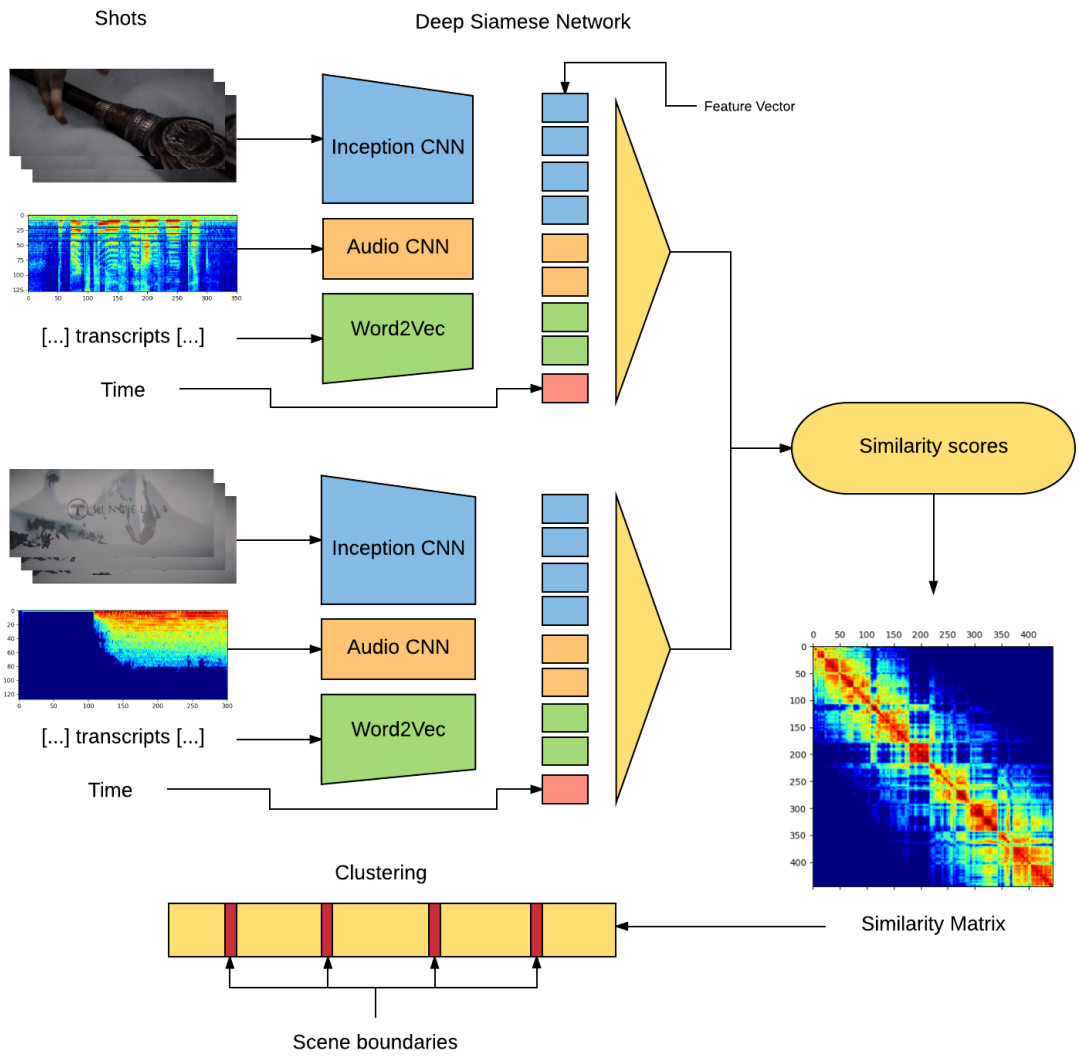

Figure 3.2: *Architecture overview of the Siamese version of the Deep Scene model. We extract visual, audible, textual features, and time of shots and concatenate all those features into a single feature vector which is being fed into a deep Siamese neural network. The network then outputs an encoding of the shot, the encodings are converted into similarity scores by calculating the cosine similarity between the pairs. Finally, adjacent shots in the similarity matrix are clustered in order to produce the final scene boundaries.*

a certain threshold the detector marks a frame as a shot boundary. Note that at training time we do not need to find the locations of the shot boundaries as this data is contained within the BBC Planet Earth dataset which we utilize for training. See Section 5.2 for details on the training set.

3.3.2 Visual features

After partitioning the video into shots we extract the visual features of the shots. For every shot we extract three frames, for the duration of the shot d and time t we extract the frames at $t = \frac{1}{3}d$, $t = \frac{1}{2}d$, and $t = \frac{2}{3}d$. The rationale behind extracting multiple frames is that a single frame is not always a good representative for the entire shot, by picking multiple frames we try to produce a sequence of frames that represent a shot well. Furthermore, this also helps to combat the impact of shots with a lot of motion blur. We did not try more advanced frame sampling strategies as recent research reported that this only has a limited effect [3].

After extracting frames for the shots we proceed by feeding each of the frames into a *convolutional neural network*. Specifically, we use a pre-trained version of Inception V3 [35]. We feed the images into the first layer of the network and then capture 2048 dimensional vectors (also referred to as *bottlenecks*) from the last fully-connected layer, just before the final softmax layer that does class predictions in Inception. This penultimate layer has been trained to output a set of values that's good enough for Inception's classifier to use to distinguish between all the classes it's been asked to recognize. That means it has to be a meaningful and compact summary of the images, since it has to contain enough information for Inception's classifier to make a good choice in a very small set of values. The reason that this bottleneck is useful for our model is that it turns out the kind of information needed to distinguish between all the 1,000 classes in ImageNet that the network was trained on is often also useful to distinguish between new kinds of objects. Our network exploits the information in this bottleneck to make decisions about the similarity of shots.

After we collected Inception's bottleneck vector for each of the frames we perform global temporal max pooling on the vectors to produce a single 2048 dimensional vector that describes the visual contents of the shot. This forms the beginning of our feature vector.

3.3.3 Transcripts

Besides the visual content of a shot, we want to take into account the contents of the transcript while still maintaining a shot based representation. Given that a shot can contain a variable number of words, and that fully-connected layers in a feed-forward neural network require a fixed size inputs, we exploit a variant of the bag-of-words approach similar to [2]. We do this by placing a context window of 20 seconds around the center of a shot. We then extract all the words within this window from the transcript. For each of the extracted words we do a K-Nearest Neighbors search with $k = 5$ on the *Google News* word2vec corpus [24]. The Google News word2vec corpus contains 300 dimensional embeddings of about 3 billion words. We average

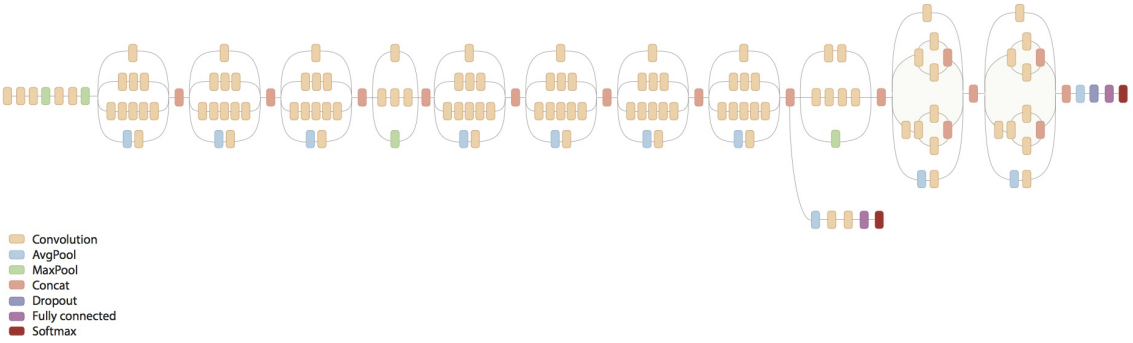

Figure 3.3: *Inception V3 Architecture. We feed the images into the first convolutional layer, and capture the output vectors from the layer just before the final softmax layer. Note that the Inception architecture is incredibly deep, while it employs only 25 million parameters.*

all the found neighbors, including the embedding for the word in the transcript, for all the words in the context window into a single descriptor. This descriptor is a 300 dimensional vector which represents the textual representation of the shot.

If transcripts are not available these could be obtained by using a speech-to-text system.

The idea here is that descriptor of the transcript for a given shot is very close in Word2Vec embedding space to a descriptor of a transcript of a shot belonging to the same scene.

3.3.4 Audio

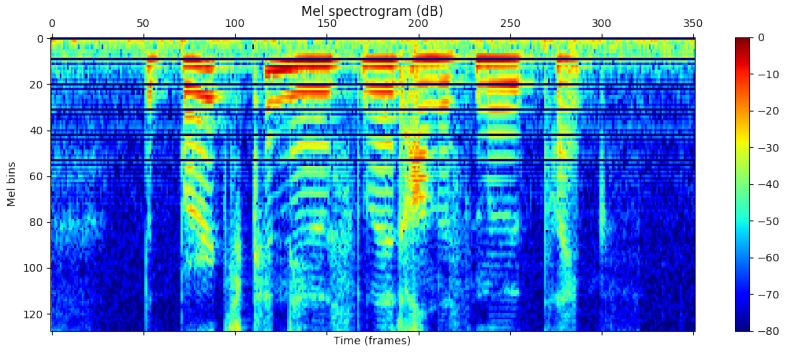

Figure 3.4: *A mel-scaled spectrogram of 3 seconds of audio from a Planet Earth episode.*

In professional video editing, audio is often used to underline the development of a story. Thus the audio of a video could also be a meaningful cue to detect scenes.

For every shot we sample the audio signal at a rate of 16000 Hz, if the signal consists of multiple channels, e.g. stereo audio, we convert it into mono by discarding

the right channel. We also tried averaging the audio channels but that did not yield any noticable improvements. We then split the audio signal in three second fragments. Then, for each of the audio fragments we perform the short-time Fourier transform (STFT) to obtain a time-frequency representation of the audio in the form of a spectrogram. For the STFT, we use a frame size of 0.025 seconds and a frame stride of 0.01 seconds. This equates to a window size of 400 samples and a window overlap of 160 samples. The linear frequency scale is then converted into a mel-frequency scale to reduce dimensionality. A mel-scale is based on the human auditory system and is approximately logarithmic above 1 kHz. We use 128 mel-frequency bins following recent research [36].

We then feed the mel-scaled spectrograms into a Convolutional Neural Network that was specifically designed for music similarity detection following practices in [12]. Our ConvNet for audio consists out of 3 convolutional layers, followed by pooling layers, ReLU's and finally two dense layers. Our ConvNet differs from traditional ConvNets used for image classification in the sense that *all* convolutions are 1 dimensional along the time dimension. Unlike images, it is important to realize that the two axes of a spectrogram have different meanings (time vs. frequency). Because of that, it does not make sense to use square filters, which is typically done in ConvNets for images.

The architecture of our Audio ConvNet is visualized in Figure 3.5 and is identical to the architecture as described in [12] where this ConvNet was employed for tag prediction. The first convolutional layer consists out of 32 kernels with a receptive field of 128 mel frequencies by 8 frames. After the first convolution we perform max pooling with a pool size of 4 frames, followed by ReLU activation. This process is then repeated two more times with the convolutional layers having a receptive field of 1x8 and 1x4 and the max pooling layer having a pooling size of 1x2, respectively. Then the feature maps are fed into two fully connected layers consisting out of 1024 and 512 neurons each that produce the final audio encoding for the sound fragment.

We feed all the audio fragments belonging to a single clip through this ConvNet and finally we perform a combination of global temporal L2, max, and average pooling in order to produce a fixed length audio descriptor for a specific shot. This descriptor is then concatenated on the vector containing the other features.

3.3.5 Time

Lastly, we also feed the normalized location of the shot in the video t into the network. Here the idea is that the network should be able to capture the relationship between the temporal distance of shots.

3.4 Loss Function

After all the features have been extracted for a pair of shots and we have two fixed feature vectors, we feed these feature vectors into a *Siamese neural network*. A Siamese neural network consists of two identical sub-networks (i.e. identical weights, layer pa-

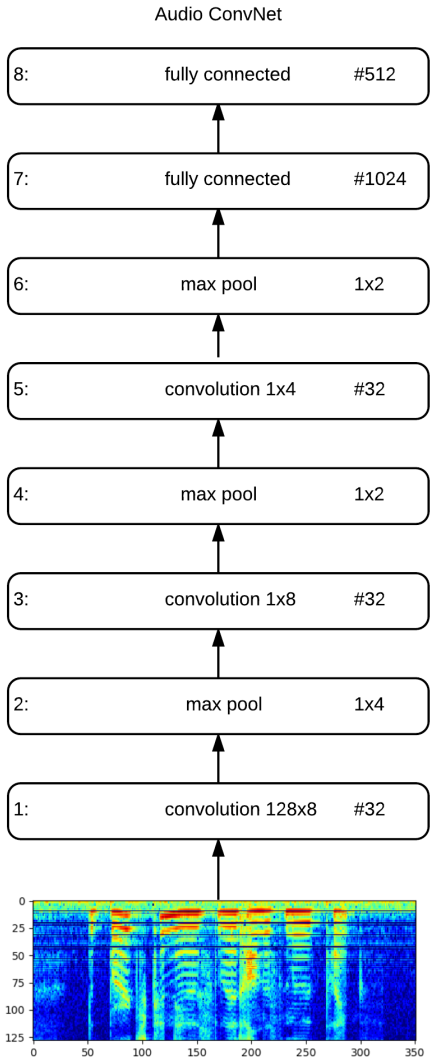

Figure 3.5: *Architecture of our Convolutional Neural Network for extracting audio features. The # operator indicates the number of neurons. ReLU's have been omitted for brevity.*

rameters, etc.) joined at their outputs. During training the two sub-networks extract features from two shots, while the joining neuron measures the distance between the two feature vectors. Our Siamese neural network consists out of two fully-connected layers with ReLU's in-between in each branch that learn how to weight the components in the feature vector to get the final similarity scores for shots.

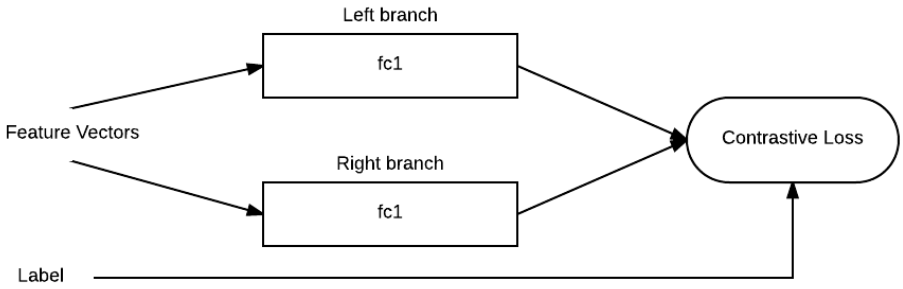

Figure 3.6: *The architecture of the Siamese Neural network used in our model. The fully-connected layers learn how to weight the components in the feature vectors in order to get the final similarity scores. At training time the network is trained using a Contrastive Loss term.*

3.4.1 Constrastive Loss

The problem of the fully-connected layers in our model is to find a function that maps high dimensional input patterns, such as our shot feature vectors, to lower dimensional outputs, given neighbourhood relationships between samples in the input space. In our case the neighbourhood relationships are the adjacent pairs of shots.

More precisely the task is to find a parametric function with the following properties:

1. Simple distance measures in the output space (such as euclidean distance) should approximate the *neighborhood relationships* in the input space. To be concrete this means that the function should map shots belonging to the same scene closer to each other than shots belonging to a different scene.

2. The mapping should not be constrained to implementing simple distance measures in the input space and should be able to learn invariances to complex transformations.

3. It should be *faithful* even for samples whose neighborhood relationships are unknown.

Such a function can be found by training the network with a contrastive loss function as introduced in [14]:

$$L(\mathbf{w}, Y, \vec{X_1}, \vec{X_2}) = (1 - Y)\frac{1}{2}(D_w)^2 + (Y)\frac{1}{2}\max(0, m - D_w)^2$$

Where $\vec{X_1}$ and $\vec{X_2}$ denote a pair of feature vectors for adjacent shots. Y is a binary label assigned to the pair of shots. $Y = 0$ denoting a pair of similar shots, *i.e* the shots belong to the same scene, and $Y = 1$ denoting dissimilar shots. m is the

contrastive loss margin and D_w is the euclidean distance between the outputs of our Siamese network G_w:

$$D_w(\vec{X_1}, \vec{X_2}) = \|G_w(\vec{X_1}) - G_w(\vec{X_2})\|_2$$

Let I be the set of training shot pairs and N be the number of pairs then the scalar loss for an entire batch of shot pairs then becomes:

$$L(\mathbf{w}, I) = \frac{1}{2N} \sum_{(\vec{X_i}, \vec{X_j}) \in I}^{N} Y_{\vec{X_i}\vec{X_j}} D_w(\vec{X_i}, \vec{X_j})^2 + (1 - Y_{\vec{X_i}\vec{X_j}}) \max(m - D_w(\vec{X_i}, \vec{X_j}), 0)^2$$

3.4.2 Triplet Loss

We also evaluated a variant on the Siamese network that uses triplets consisting out of an anchor (a randomly picked shot), positive and negative exemplar for training rather than a pair of exemplars and a label like the Siamese network.

The objective of triplet loss [31] is to ensure that a shot $\vec{X_i^a}$ (anchor) of a specific scene is closer to all other shots $\vec{X_i^p}$ (positive) of the same scene than it is to any shot $\vec{X_i^n}$ (negative) than the shots of any other scene.

Thus we want,

$$\|G_w(\vec{X_i^a}) - G_w(\vec{X_i^p})\|_2^2 + \alpha < \|G_w(\vec{X_i^a}) - G_w(\vec{X_i^n})\|_2^2,$$

$$\forall (G_w(\vec{X_i^a}), G_w(\vec{X_i^p}), G_w(\vec{X_i^n})) \in \tau.$$

where α is a margin that is enforced between positive and negative pairs. τ is the set of all possible triplets in the training set and has cardinality N. Note that G_w in this case denotes the output of the triplet network, rather than the Siamese network. We are using the same symbol, since in practice the implementation of the network is identical to the Siamese network.

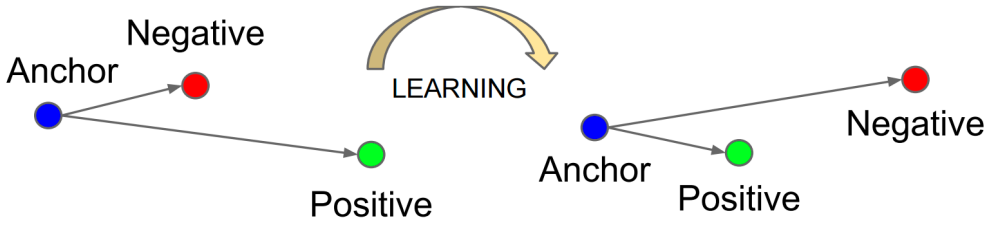

Figure 3.7: *The* ***Triplet Loss*** *minimizes the distance between an anchor and a positive, both of which have the same identity, and maximizes the distance between the anchor and a negative of a different identity. Image adapted from [31].*

The loss being mininimized is then:

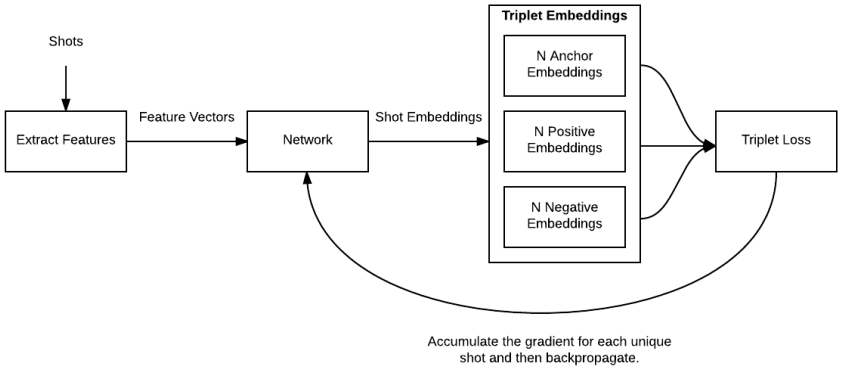

Figure 3.8: *Diagram depicting the architecture of the network at training time. Embeddings for shots are computed individually, triplets are formed based on the embeddings and used to compute the Triplet loss.*

$$L(\mathbf{w}, N) = \frac{1}{N}\sum_{i}^{N} \max(0, \|G_w(\vec{X_i^a}) - G_w(\vec{X_i^p})\|_2^2 + \alpha < \|G_w(\vec{X_i^a}) - G_w(\vec{X_i^n})\|_2^2)$$

3.4.3 Regularization

We use a squared l_2-norm for regularization in order to prevent the network from overfitting on our training dataset, leading to the complete learning objective function for the Siamese architecture:

$$L(\mathbf{w}, I) = \frac{\lambda}{2}\|\mathbf{w}\|_2^2 + \frac{1}{2N}\sum_{(\vec{X}_i, \vec{X}_j) \in I}^{N} Y_{\vec{X}_i \vec{X}_j} D_w(\vec{X}_i, \vec{X}_j)^2 + (1 - Y_{\vec{X}_i \vec{X}_j}) \max(m - D_w(\vec{X}_i, \vec{X}_j), 0)^2$$

where N denotes the number of training pairs, I the set of training pairs and $\mathbf{w}$ the weights of the neural network.

For the triplet architecture the complete learning objective then becomes:

$$L(\mathbf{w}, N) = \frac{\lambda}{2}\|\mathbf{w}\|_2^2 + \frac{1}{N}\sum_{i}^{N} \max(0, \|G_w(\vec{X_i^a}) - G_w(\vec{X_i^p})\|_2^2 + \alpha < \|G_w(\vec{X_i^a}) - G_w(\vec{X_i^n})\|_2^2)$$

Where N in this case denotes the cardinality of the training set. Again, $\mathbf{w}$ represents the weights of the neural network.

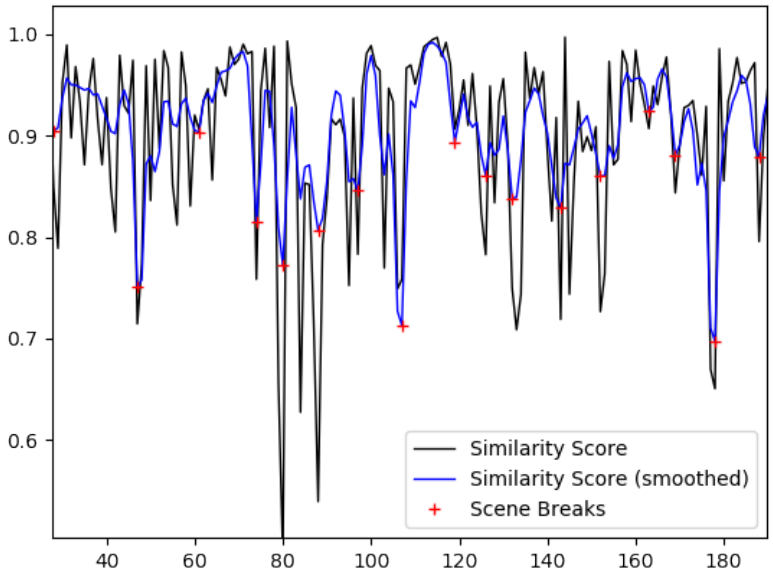

Figure 3.9: *Plot of the similarity scores for adjacent pairs of shots. The black line denotes the raw similarity scores and the blue line the smoothed similarity scores. Scene breaks correspond to local minima in the similarity score plot and are denoted by a red +.*

3.5 Clustering

After obtaining embeddings for all the shots in the set of pairs I we need to segment the sequence of shots into scenes. We do this by posing segmentation as a clustering problem.

First we convert the embeddings into similarity scores by calculating the cosine similarity between all the shot embeddings. The similarity scores are then stored in a matrix S, which we call the *similarity matrix*. The value in the matrix at S_{ij} indicates the similarity score between the i-th and the j-th shot.

Because we also want to emphasize the temporal distance between shots, we also construct temporal affinity matrix T. The temporal affinity between shots is calculated by subtracting the normalized position of shot i in the video by the normalized position of shot j. We also make sure that the temporal distance matrix is symmetric by setting T_{ji} to T_{ij}, thus T_{ij} indicates the temporal distance between shot i and j.

Subsequently we combine these two matrices in order to produce a final similarity matrix on which we perform clustering by computing $C_{sim} = S - 2T$.

We then apply a multidimensional Gaussian filter to the similarity matrix C in order to smooth the similarity scores. After which we construct a vector containing the similarity scores along the diagonal of the matrix starting with shot $[C_{01}, C_{12}, C_{23}, ...C_{n-1,n}]$ i.e. this vector contains the similarity scores for all the adjacent shots in a video.

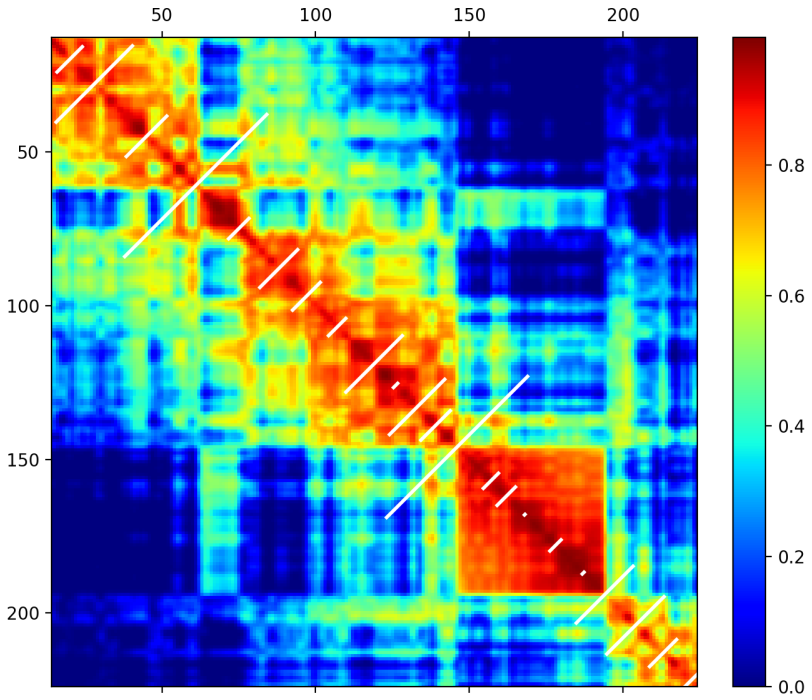

Figure 3.10: *Fragment of the similarity matrix for the shots of the first episode of Planet Earth. Scene boundaries are placed in areas of low similarity and are indicated by the white diagonal lines. The size of the white diagonal line indicates the model's confidence in whether a scene break should be placed.*

Next, we try to find the local minima in this vector of similarity scores, drops in similarity score indicate that shots differ significantly from each other, which may indicate a change in story or subject. Thus we place the final scene boundaries at these local minima (see Figure 3.9). In order to prevent oversegmentation we constrain local minima to be at least 4 data points (shots) apart.

Finally, we determine the model's confidence in the placed scene breaks. We do this by calculating a confidence score based on the gradient of the similarity scores for the adjacent pairs of shots. We compute the gradient using second order accurate central differences in the interior and either first differences or second order accurate one-sides (forward or backwards) differences at the boundaries. We then look at the gradient for the shots where we placed a scene break and take the absolute value of the gradient. A high value for the gradient indicates a bigger gap in similarity scores, and thus it is more likely that a scene break occurred. In Figure 3.10 the relative difference between confidence scores is demonstrated by the length of the white diagonal line.

We also tried other algorithms for clustering such as K-means, TWSS clustering [3], and Spectral Clustering [39][2], and maximizing Newman's Modularity Measure [25], but we found the approach described above to perform best for our use-case.

Chapter 4

Implementation

In this chapter we will describe how the model as described in the previous chapter has been implemented. We will describe which frameworks we have used and why we have used them. Furthermore we will shed some light on the demo we created that we used to subjectively evaluate the performance of the model and to demonstrate progress and performance to stakeholders.

4.1 TensorFlow

We implemented the model as described in Chapter 3 using TensorFlow. TensorFlow [1] is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (which we call tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. TensorFlow was originally developed by researchers and engineers working on the Google Brain Team with Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well. TensorFlow exposes APIs in multiple programming languages, but currently only the Python API can be considered stable.

We choose to implement our model using TensorFlow for a number of reasons, of which the most important one was that there is a pre-trained Inception V3 model available for TensorFlow, which we rely on for our visual features. Training convolutional neural networks from scratch is an expensive task in terms of resources, both in time and in computing resources. Having a pre-trained model available saved us at least a week in terms of development time.

Furthermore, TensorFlow is also used in a production environment at JW Player which meant that there was knowledge available within the company.

4.2 Microservice Architecture

We modeled our scene detection pipeline using a microservice architecture. The idea of a microservice architecture is that you structure applications as a collection of loosely coupled services, where each service has it's own responsibility. The microservice architecture enables the continuous delivery/deployment of large, complex applications.

Compared to a monolithic architecture, microservices also often allow for more flexible scaling of applications - it allows you to identify bottlenecks and scale those separately from the rest of the services.

Microservices however, are not a silver bullet. There are a number of drawbacks associated to them: deployment of microservices can be complex because of the dependencies associated with them, you may have to coordinate among multiple services, which may not be as straightforward as deploying a single monolith. As a side effect of that, testing of microservices can also be complex.

For that reason we split our scene detection pipeline up into just two microservices during development, a *demo* service and a *api* service. The microservices were packaged in *Docker* containers to allow for easy deployment and we used *Docker Compose* for container orchestration.

4.3 API

The API microservice encapsulates the entire scene detection pipeline, and exposes a single endpoint *"scene_detector"* to which scene detection jobs can be submitted by sending a HTTP POST request. Scene detection jobs require three JSON-encoded parameters in their body: an URL to a video, an (optional) URL to a transcript and a callback URL. The callback URL is required because in general scene detection jobs are long-running tasks, and we do not want to keep an HTTP connection open, thus the API will execute a *GET* request to the callback URL whenever it has an update to report about the scene detection job.

A sample scene detection request looks like the following:

```
POST /scene_detector HTTP/1.1
Content-Type: application/json

{
  "video_url": "https://host.com/video.mp4",
  "transcript_url": "https://host.com/transcript.srt",
  "callback_url": "http://localhost:8081/callback"
}
```

After submitting a request the API will return HTTP status code 200, if the request was valid and successfully appended to the job queue.

As soon as the queue contains any jobs the *SceneDetectionWorker* running inside the API container, will take a job from the queue and start processing it. First it will

analyze the video for shot cuts, then it will extract frames for every shot as described in 3.3.2. After extracting frames, the transcripts will be processed as described in Chapter 3.3.3 and the audio will be processed as described in Chapter 3.3.4. Finally, the built feature vectors will be fed into our Siamese (or triplet) neural network and clustering will be performed, after which we will return the following response to the client:

```
POST /callback HTTP/1.0
Content-Type: application/json

{
    "video": "https://host.com/video.mp4",
    "frame_rate": 24.0,
    "shots": [1018, 1035, 1056, 1078, 1100, ...],
    "scene_breaks": [0, 6, 23, 37, 42, ...]
}
```

The response contains 4 fields:

- **video**: The URL of the video that was processed.
- **frame_rate**: The frame rate of the original video.
- **shots**: The indexes of the frames in the video where we found a shot cut.
- **scene_breaks**: The indexes of the shots where we detected a scene break.

4.4 Demo

Since our scene detection pipeline only exposes a single low-level HTTP endpoint that only supports JSON we've also built a demo application that allows you to process a video from the browser.

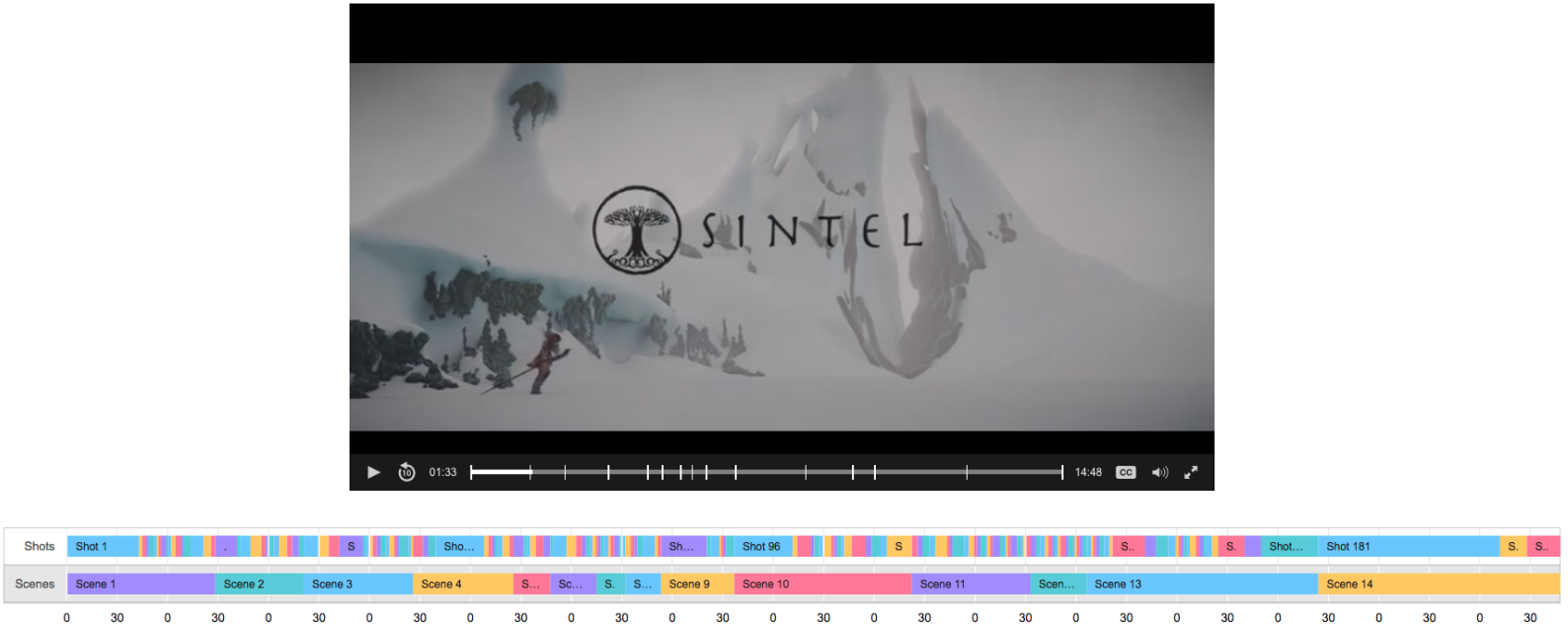

Figure 4.1: *Screenshot of the Deep Scene demo page. Scene breaks are indicated by white dashes on the seek bar of the player. The timeline below the player depicts the location of shots and the location of scene breaks in the upper, and bottom row respectively.*

The demo exposes a web application that exposes two input fields, one input field for an arbitrary video URL and one input field for an URL to a subtitle, which is optional. Furthermore a setup button is exposed. After entering a video url, a subtitle url and hitting setup a scene detection job will be submitted to the queue. The request is made over a WebSocket to the backend of the demo, which will forward the request to the Deep Scene API. We need this intermediate layer because the Deep Scene API requires a callback URL, and thus we can not directly send the result back to a client's browser as the browser does not expose an HTTP server. The backend of the demo serves this purpose by exposing a callback endpoint, after the scene detection job is done the demo will cache the results of the scene detection job and push the results over a WebSocket connection to the client of the demo.

As soon as the client receives data from the demo backend, a timeline will be initialized and scene breaks will be plotted on the seekbar of the player present on the page.

The demo page will also initialise a video player that plays midroll advertisements. The midroll advertisements are scheduled using a simple heuristic that places the ads at the three most confident scene breaks. Interestingly, we observed that in general the most confident scene breaks are often well spread out across the video.

4.5 Python Libraries

Besides TensorFlow we also used a number of other Python libraries, notably:

- NumPy: NumPy is a fundamental package for scientific computing, it offers: powerful N-dimensional array objects, sophisticated (broadcasting) functions and useuful linear algebra, Fourier transform and random number capabilities.
- SciPy: The SciPy library is another Python package for scientific computing, it relies heavily on NumPy and offers efficient routines for image and signal processing, statistics etc.
- scikit-learn: scikit-learn is a machine learning library for Python, it is built on top of NumPy, SciPy and matplotlib. It contains utilities for solving classification, regression, clustering, dimensionality reduction, model selection and preprocessing problems.
- PySceneDetect: We use the PySceneDetect package for finding shot cuts in videos, which we need to do at inference time. PySceneDetect relies heavily on OpenCV and ffmpeg.
- pysrt: pysrt is a Python parser for SubRip (srt) files, we use it to parse and slice transcripts when doing feature preprocessing.
- gensim: We use gensim [27] for finding similar words in the Google News Word2Vec corpus. gensim offers a clean API for finding similar words and doing k-NN searches.

- moviepy: moviepy is a Python wrapper around ffmpeg, we use it to extract frames and audio from video containers.
- librosa: librosa is a Python library for audio and music analysis, we use it to convert audio into mel-scaled spectrograms.
- matplotlib: We rely heavily on matplotlib for visualizing data and to apply color to the audio spectrograms.
- Jupyter Notebooks (IPython): Jupyter Notebooks were heavily used when prototyping, and trying out new ideas.
- Pillow: Pillow is an image library for Python, it is used to convert raw pixels into JPEG and PNG images.
- Falcon: Falcon is a minimalist, very fast Python web framework for building microservices, app backends and higher level frameworks. We use it to power the Deep Scene API.
- Flask: Flask was used as web framework in the demo application of Deep Scene, we used Flask over Falcon because of it's WebSocket capabilities.

4.6 Deep Scene at scale

The current architecture we're using in Deep Scene is not ideal for a production setting. We noticed that both the detection of scene cuts, and the extraction of frames is quite an expensive task in terms of computing resources. Ideally, we would detect scene cuts, extract frames and calculate mel-spectrograms during the transcoding of videos in JW Player's transcoding pipeline. In such a case, we would only ever have to process a video once.

Unfortunately, integrating all those tasks in the transcoding pipeline is quite a complicated process. Alternatively, it would already greatly help if we would split out the detection of scene cuts and, frame extractions to their own microservices. The microservices could then write the frames and scene cut data to some form of shared storage (such as Amazon S3). After those services are done with processing a video, they could signal the container employing the neural network that all the dependencies are ready, after which we can run inference on the data. This would allow for better scalability and horizontal scaling of services across the entire pipeline.

Chapter 5

Results

5.1 Performance metrics

The problem of measuring scene detection performance is significantly different from that of measuring shot detection performance. Classical boundary detection scores, such as Precision and Recall, fail to convey the true perception of an error, which is different for an off-by-one shot or for a completely missed scene boundary.

In [38] the Coverage and Overflow measures where proposed to combat this problem.

Coverage C measures the quantity of shots belonging to the same scene correctly grouped together, while Overflow O measures to what extent shots not belonging to the same scene are erroneously grouped together. Formally, given the set of automatically detected scenes $s = [s_1, s_2, s_3, ..., s_m]$, and the ground truth scenes $\tilde{s} = [\tilde{s}_1, \tilde{s}_2, \tilde{s}_3, ..., \tilde{s}_n]$, where each element of s and $\tilde{s}$ is a set of shot indexes, the coverage C_t of scene $\tilde{S}_t$ is defined as:

$$C_t = \frac{\max_{i=1...,m} \#(s_i \cap \tilde{s}_t)}{\#(\tilde{s}_t)}$$

where the operator # counts the amount of shots s_i or s_t contains.

The overflow of a scene $\tilde{s}_t$, O_t, is the amount of overlap of every s_i corresponding to $\tilde{s}_t$ with the two surrounding scenes:

$$O_t = \frac{\sum_{i=1}^{m} \#(s_i \setminus \tilde{s}_t) \cdot \min(1, \#(s_i \cap \tilde{s}_t))}{\#(\tilde{s}_t - 1) + \#(\tilde{s}_t + 1)}$$

The per-ground-truth-scene measures are aggregated for the entire video by averaging, weighting them by the number of shots in each scene.

As noted in [38] there are three important applications for these measurements. First of all, they are useful to compare the performance of individual features. Second, the measurements show to what extent segmentation of a video sequence is theoretically possible, i.e., under ideal circumstances. The ideal feature/threshold combination has $C = 1$ and $O = 0$. The difference between the actual measurements and the ideal is the inherent complexity of the segmentation problem. Third, when

coverage and overflow are plotted against one another, an appropriate threshold can be selected depending on the user's preferences for amount of overflow (undersegmentation) and coverage (oversegmentation).

For our specific use case we prefer to slightly undersegment the video in order to allow for ad break insertions in clearly distinct scenes.

Unfortunately, these measures have a number of drawbacks. As noted in [2] and [32] the relation of O with the previous and next scenes create unreasonable dependencies between an error and the length of a scene observed many shots before it. Moreover, C only depends on the maximum overlapping scene, and does not penalize the other overlapping scenes in any way: any over-segmentation in the other overlapping scenes does not change the measure value.

Hence in [2] the symmetric *maximum intersection-over-union* metric denoted as M_{iou} is proposed to assess the quality of detected scenes. For each ground-truth scene, we take the maximum intersection-over-union with the detected scenes, averaging them on the whole video. Then the same is done for detected scenes against ground-truth scenes, and the two quantities are again averaged. An important note is that both intersection and union are measured in terms of frame lengths for the shots, thus weighting the shots with their relative significance. The final measure is thus given by:

$$M_{iou} = \frac{1}{2}\left(\frac{1}{n}\sum_{i=1}^{n}\max_{j\in\mathbb{N}_m}\frac{\tilde{s}_i \cap s_j}{\tilde{s}_i \cup \tilde{s}_j} + \frac{1}{m}\sum_{j=1}^{m}\max_{m\in\mathbb{N}_n}\frac{\tilde{s}_i \cap s_j}{\tilde{s}_i \cup \tilde{s}_j}\right)$$

5.2 Experiments setting

Similar to [2] we evaluate the performance of our solution on 11 episodes from the BBC TV series *Planet Earth*. Each episode of Planet Earth is approximately 50 minutes long. The entire dataset contains annotations for around 4500 shots and 700 scenes. Shots and scenes of the entire dataset have been manually annotated by a set of human experts.

To train our model, we employ Adam [20] which we initialize with a learning rate of $\alpha = 0.1$. We decay the learning rate every 100 steps with a decay rate of 0.96. For Adam we use $\beta_1 = 0.9$, $\beta_2 = 0.999$, and we use $\epsilon = 1e-8$ for numerical stability. We use a weight decay of $\lambda = 0.0005$ for the weights in the Audio ConvNet and the last fully-connected layers. Furthermore we found that a contrastive loss margin of $m = 3.14$ for the Siamese network and a margin $\alpha = 0.2$ for the Triplet network, and $\sigma = 1$ for the Gaussian filter yielded the best clustering results.

5.3 Evaluation

We compare the triplet version of our model against Baraldi's Deep Neural Network [2], which is a model that is similar to ours, but with a couple of notable differences.

In [2] only visual and textual features are taken into account. In [2] CaffeNet is used as as CNN, which is a variation on the well-known AlexNet [22]. We are using Inception, which is deeper and more accurate on the task of image classification than AlexNet. We did not compare our model against [3] since that model relies heavily on semantic features that require transcripts to be available. In our case transcripts are often not available, thus we want to limit our dependency on transcripts and only consider models that employ perceptual features.

During evaluation we used the ground-truth data for shots, since the performance of the shot detector affects scene detection performance.

Furthermore for completeness we also included the scene detection performance of the Scene Transition Graph and the Needleman-Wunsch based algorithms.

Episode	STG [33]	NW [10]	SDN [2]	Our Triplet Network
From Pole to Pole	0.42	0.35	**0.50**	0.49
Mountains	0.40	0.31	0.53	**0.58**
Fresh Water	0.39	0.34	**0.52**	0.51
Caves	0.37	0.33	**0.55**	**0.55**
Deserts	0.36	0.33	0.36	**0.44**
Ice Worlds	0.39	0.37	**0.51**	0.50
Great Plains	0.46	0.37	0.47	**0.48**
Jungles	0.45	0.38	**0.51**	0.45
Shallow Seas	0.46	0.32	**0.51**	0.43
Seasonal Forests	0.42	0.20	0.38	**0.44**
Ocean Deep	0.34	0.36	**0.48**	0.46
Average	0.41	0.33	0.48	0.48

Table 5.1: *Scene Detection performance on the individual episodes from BBC Planet Earth.*

According to the table our model performs similar to Baraldi's Deep Siamese Neural Network on most episodes, but also significantly better on some episodes and significantly worse on other episodes. It is hard to tell where those differences exactly come from. Our suspicion is that the biggest differences lie in the clustering algorithm. We observed that our clustering algorithm is far from perfect: the main problems are that due to the smoothing of the similarity signal it is not always capable of detecting scene breaks which manifest themselves very subtle in the signal of similarity scores and at this moment the algorithm is also artificially constrained to detect scenes that contain at least 4 shots. We think that maybe a neural network can be employed for the clustering step instead, and we are currently evaluating such an approach.

Chapter 6

Development Process

In this chapter we will shed a light on the development process of Deep Scene in general.

6.1 Version Control System

As version control system for Deep Scene *git* was used. We choose to use git because git is fast, distributed and nowadays the industry standard. As remote repository we used a private GitHub repository under the JW Player organization. Using GitHub was a company requirement. Furthermore we also conducted code reviews through GitHub's Pull Request Review feature.

6.2 Bugtracking

During the project we used Atlassian's JIRA software as issue tracker and as planning tool. Tickets were made for approaches and experiments that were performed and tracked in those tickets.

6.3 Communication

In the first phase of the Deep Scene project daily sync-ups were held to discuss progress, approaches and the results of experiments. In general we would sync-up by using Google Hangouts' video call feature, but often we would also sync-up by using asynchronous communication tools such as HipChat and Slack.

6.4 Project agenda

Initially we outlined a very rough project planning (see the project plan document), but due the amount of uncertainty whether (parts of) this project would be successful we deviated from this planning very quickly. For example, initially we planned to evaluate a Recurrent Neural Network based solution, but during literature research

we realised that it would be better if we could solve this problem using a standard feed-forward neural network because others have reported better results with standard neural networks than with Recurrent Neural Networks for solving related problems. Furthermore our dataset is also rather sparse and standard feed-forward networks have better performance properties.

From that point moving forward we decided that our process would be roughly the following:

1. Identify a bottleneck in our current architecture.
2. Perform literature research with regard to the bottleneck, has this problem been solved before? Take notes.
3. Once we are comfortable with our knowledge around the area of the bottleneck, decide on a potential solution for the problem.
4. Implement that solution.
5. Analyse the given solution, did the situation improve?

It turned out that since we're dealing with neural networks, especially the analysis phase was very difficult. For example, often it appeared that changes made the similarity matrix look "better", but at the same time we could not measure a performance improvement. Even with some of the optimisation's we did under the hood, the network took a long time to train. Finding the correct configuration of hyper parameters often took more time than actually writing code for a new part of the model

An overview which task was completed when can be found in Appendix A.

Chapter 7

Conclusion

In this work we presented an end-to-end pipeline for the detection of scenes in broadcast videos. We have demonstrated how both visual, audible and textual features extracted from the shots in a video can be used to learn a distance metric between shots that can be leveraged as a measure for semantic coherency. We presented two architectures for a Deep Neural Network which is capable of learning this distance metric: a Siamese version of the network and a triplet version of the network. We explained how this distance measure between shots can be used to partition a video into meaningful and story-telling parts. Furthermore we demonstrated the effectiveness of our Triplet Network by comparing it to other scene detection models using the maximum intersection-over-union measure.

7.1 Future Work

We think there is still a lot of room for performance improvement of this model. For example it would be interesting to build a Recurrent Neural Network version of this model that is capable of memorising what it has "seen", and is capable of using that to build relations between shots. It would also be interesting to see the effect of 3D convolutions, we're thinking these might allow the network to better classify locations and objects in sets of frames, which can then again be used to draw better inferences between shots. Furthermore we think there is a lot of improvement possible in the clustering stage of the model. Our current clustering algorithm constrains scenes to have a minimum length of 4 shots, which in a real world scenario is not per se correct. We also think that training the network on a bigger dataset will have a positive impact on the scene detection performance.

Chapter 8

Reflection

Developing the scene detection model turned out to be a very challenging, but interesting task. During the development of Deep Scene I acquired an incredible amount of knowledge of all sorts of aspects that play a role with Deep Learning based models. In fact I often caught myself starting to read with one paper, and then going from paper to paper just to get a better understanding of the problem and its solution. To give some examples, this is how I learned about concepts such as *batch normalization*, *spatial pyramid pooling* and *deconvolutions*, just to name a few. In general I am quite happy with the result, a framework for end-to-end scene detection, that performs competitively to Baraldi's similar model. I am looking forward to see what the impact of this model will be when its being used to determine the best location of an ad break in video.

Appendix A

Task Overview

Eventually features have been implemented in somewhat the following order. The list below is based on the git commit history of the Deep Scene repository. It does not include efforts with regard to hyper-parameter tuning or ideas that were evaluated in IPython notebooks and were shot down, for whichever reason, before they made it into the actual code base. Literature research is also not accounted for in this list as that essentially was a continuously ongoing effort. This means that this is not a complete list by any means of work that was done but merely gives an indication of what the focus was on and in which order. Note that every change to the model was paired with literature research, retraining the model and a lot of tinkering with hyper-parameters. See Chapter 6 for details on the development process.

February 6 - Mar 10	Implemented Baraldi's network in TensorFlow based on the description in his paper [2].
February 13	Implemented scripts that parse the BBC Planet Earth dataset.
February 14	Implemented scripts that capture frames from video.
February 15	Implemented caching of bottlenecks in order to prevent expensive re-computation.
February 15	Wrote instructions on how to train the model on the BBC dataset.
February 16	Introduced L2 regularization to Baraldi's model.
February 16 - March 1	Implemented clustering based on a Radial Basis Function (RBF) Kernel and Spectral Clustering given an incomplete description on how clustering should be done in Baraldi's paper.
March 2	Improved data pre-processing steps and wrote logic to artificially generate more training shot pairs. Implemented logic to balance training batches.
March 2 - 8	Fixed bugs in the loss function, added logging and TensorBoard summaries. Improved data set loading. Added support for variable batch sizes.

March 9	Implemented better balancing of training batches (dissimilar shot pairs between episodes etc.) as an effort to make the network generalize better.
March 10	Implemented support for transcripts in the preprocessor.
March 10	Implemented word embeddings as feature.
March 10 - March 13	Implemented clustering based on local minima in the similarity matrix (and tried various other approaches).
March 14 - March 15	Implemented performance evaluation metrics to evaluate model performance using IPython notebooks.
March 16 - 18	Implemented a REST HTTP API for the Deep Scene model.
March 16	Containerized the Deep Scene API.
March 17 - 21	Implemented Deep Scene Demo.
March 21	Took care of ensuring the code base is Flake8 compliant.
March 22	Added the timeline to the Deep Scene demo page.
March 23 - 27	Tracked down and fixed several bugs in the demo.
March 27	Added support for Docker compose. Created OpenCV 3.2.0 Ubuntu based Docker container with support for Python 3 and ffmpeg bindings.
March 28 - April 7	Start implementation of audio as a feature, research possible approaches. Evaluate using a ConvNet and Spatial Pyramid Pooling [17] in order to pool variable length audio into fixed length descriptors.
April 7 - April 14	Implemented a "what", "where" convolutional autoencoder [40] for Audio spectrograms.
April 14 - 17	Made various changes to the Audio AE.
April 17 - 24	Implemented another ConvNet architecture for audio based on 1D convolutions and global temporal pooling and switched to that.
April 24 - 26	Implemented Triplet Loss.
April 26 - May 1	Assigned confidence scores to scene breaks, and updated the demo respectively.
May 2	Fixed various bugs, implemented a decaying learning rate.
May 2 - 3	Experimented with various clustering strategies.
May 3 - 8	Refactored the Triplet Network to only use one single branch for the forward and backward pass by unstacking Triplets before calculating the loss similar to OpenFace. Looked at better Triplet sampling strategies.
May 8	Re-evaluated our Siamese Network vs. the Triplet variant.
May 8 - 11	Implemented L2 temporal pooling for audio.

May 12	Fixed various bugs and improved the demo.
May 12 - 17	More experimenting with different clustering strategies, evaluated Newman's measure of optimal modularity.
May 18	Presented Deep Scene at JW Insights in New York City.
May 19 - June 1	Evaluated TWSS clustering.
June 1 - June 12	Worked on implementing a clustering strategy based on a separate neural network for clustering (not yet finished).

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[2] Lorenzo Baraldi, Costantino Grana, and Rita Cucchiara. A deep siamese network for scene detection in broadcast videos. In *Proceedings of the 23rd ACM International Conference on Multimedia*, MM '15, pages 1199–1202, New York, NY, USA, 2015. ACM.

[3] Lorenzo Baraldi, Costantino Grana, and Rita Cucchiara. Recognizing and presenting the storytelling video structure with deep multimodal networks. *CoRR*, abs/1610.01376, 2016.

[4] Yoshua Bengio. Learning deep architectures for ai. *Found. Trends Mach. Learn.*, 2(1):1–127, January 2009.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[6] J.M. Boggs and D. W. Petrie. *The art of watching films*. Mayfield, 2000.

[7] R. M. Bolle, B.-L. Yeo, and M. M. Yeung. Video query: Research directions. *IBM J. Res. Dev.*, 42(2):233–252, March 1998.

[8] Jane Bromley, Isabelle Guyon, Yann Lecun, Eduard Sckinger, and Roopak Shah. Signature verification using a "siamese" time delay neural network. In *In NIPS Proc*, 1994.

[9] R. Brunelli, O. Mich, and C.M. Modena. A survey on the automatic indexing of video data,. *J. Vis. Comun. Image Represent.*, 10(2):78–112, June 1999.

[10] V. T. Chasanis, A. C. Likas, and N. P. Galatsanos. Scene detection in videos using shot clustering and sequence alignment. *IEEE Transactions on Multimedia*, 11(1):89–100, Jan 2009.

[11] Li Deng and Dong Yu. Deep learning: Methods and applications. Technical report, May 2014.

[12] Sander Dieleman and Benjamin Schrauwen. End-to-end learning for music audio. In *International Conference on Acoustics Speech and Signal Processing ICASSP*, pages 6964–6968. IEEE, 2014.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[14] Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2*, CVPR '06, pages 1735–1742, Washington, DC, USA, 2006. IEEE Computer Society.

[15] Jun Han and Claudio Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *Proceedings of the International Workshop on Artificial Neural Networks: From Natural to Artificial Neural Computation*, IWANN '96, pages 195–201, London, UK, UK, 1995. Springer-Verlag.

[16] Alan Hanjalic, R.L. Lagendijk, and Jan Biemond. Automated high-level movie segmentation for advanced video-retrieval systems. *IEEE Trans. on Circuits and Systems for Video Technology*, 9(4):580–588, 1999.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *CoRR*, abs/1406.4729, 2014.

[18] Elad Hoffer and Nir Ailon. Deep metric learning using triplet network. *CoRR*, abs/1412.6622, 2014.

[19] A Karpathy. CS231n: Convolutional neural networks for visual recognition. `http://cs231n.github.io/`, 2016. Accessed: 2017-04-10.

[20] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[21] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. Siamese neural networks for one-shot image recognition. 2015.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[23] Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y. Ng. Unsupervised learning of hierarchical representations with convolutional deep belief networks. *Commun. ACM*, 54(10):95–103, October 2011.

[24] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[25] M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006.

[26] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[27] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[28] Yong Rui, Thomas S. Huang, and Sharad Mehrotra. Constructing table-of-content for videos. *Multimedia Systems*, 7(5):359–368, 1999.

[29] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

[30] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[31] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. *CoRR*, abs/1503.03832, 2015.

[32] P. Sidiropoulos, V. Mezaris, I. Kompatsiaris, and J. Kittler. Differential edit distance: A metric for scene segmentation evaluation. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(6):904–914, June 2012.

[33] Panagiotis Sidiropoulos, Vasileios Mezaris, Ioannis Kompatsiaris, Hugo Meinedo, and Isabel Trancoso. Multi-modal scene segmentation using scene transition graphs. In *Proceedings of the 17th ACM International Conference on Multimedia*, MM '09, pages 665–668, New York, NY, USA, 2009. ACM.

[34] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[35] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

[36] Aaron van den Oord, Sander Dieleman, and Benjamin Schrauwen. Deep content-based music recommendation. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2643–2651. Curran Associates, Inc., 2013.

[37] Rahul Rama Varior, Mrinal Haloi, and Gang Wang. Gated siamese convolutional neural network architecture for human re-identification. *CoRR*, abs/1607.08378, 2016.

[38] J. Vendrig and M. Worring. Systematic evaluation of logical story unit segmentation. *IEEE Transactions on Multimedia*, 4(4):492–499, 2002.

[39] Ulrike von Luxburg. A tutorial on spectral clustering. *CoRR*, abs/0711.0189, 2007.

[40] J. Zhao, M. Mathieu, R. Goroshin, and Y. LeCun. Stacked What-Where Auto-encoders. *ArXiv e-prints*, June 2015.