VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Minh Hoang

DEVELOPING A VIDEO SUMMARIZING TOOL USING MACHINE LEARNING

Technology and Communication
2022

ACKNOWLEDGEMENTS

First of all, I would like to thank all of my teachers at VAMK, especially Dr. Ghodrat Moghadampour, my thesis report supervisor. Without his help throughout my journey at VAMK, I might not have graduated with the highest possible grades, despite having learned so much from his lectures.

Furthermore, I would like to express my gratitude to Mr. HO Tuong Vinh and Mr. LAM Quang Tung for providing me with the opportunity to work on this project and polishing up my research skills. These could be beneficial to me as I prepare for a new chapter in my career.

Finally, I would like to thank my parents in Vietnam, as well as my friends in Vaasa, in Vietnam, and the United States, for always having my back and supporting me not only financially, but also with motivation and encouragement.

Ngoc Minh, Hoang
Vaasa, Finland
11.03.2022

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Minh Hoang
Title	Developing a Video Summarizing Tool using Machine Learning
Year	2022
Language	English
Pages	49
Name of Supervisor	Ghodrat Moghadampour

In recent years, video has become a highly significant form of visual data, and the explosion of short video platforms like TikTok, Instagram, and Facebook has led people to prefer to consume short content than watching a long video without skipping any second. To solve this problem, a video summarizing system was developed in this work, to compress a long video into a much shorter version, but still preserving the important contents of the original one. As a trend in the world nowadays, Artificial Intelligence was used to make the process more efficient, and it proved to be perfectly suitable for handling this task too.

The primary goal of the project was to do research in the field of video summarization and develop a tool that can assist people in improving video summarization workflow by significantly reducing processing time and simplifying video summarizing process. The original research focused on improving the performance of the method but not on the usability for normal users.

With the application developed in this project, the user can summarize a video just by inputting the original video file and getting a summarized version of it in terms of seconds.

Keywords Artificial intelligence, machine learning, computer vision, video summarization, and multimedia processing

CONTENTS

ABSTRACT

1 INTRODUCTION 9

1.1 Background and Motivation 9

1.2 Objectives.... 10

2 THEORETICAL REVIEW 11

2.1 Artificial Intelligence 11

2.2 Machine Learning.... 12

2.3 Neural Networks and Deep Learning.... 13

2.4 Convolutional Neural Network 15

2.5 ResNet.... 16

2.6 Long Short-Term Memory.... 18

2.7 Generative Adversarial Network 20

2.8 Variational Autoencoder.... 21

2.9 Actor-Critic Model.... 22

2.10 Video Structure 23

2.11 Knapsack Algorithm 24

3 TOOLS AND TECHNOLOGIES IN USE 26

3.1 Python 26

3.2 PyTorch 26

3.3 OpenCV 27

3.4 Decord.... 27

3.5 Pydub 28

4 APPLICATION DESCRIPTION.... 30

5 IMPLEMENTATION.... 34

5.1 Training 34

5.1.1 Training Encoder 34

5.1.2 Training Decoder.... 35

5.1.3 Training Linear Compression and Discriminator.... 36

5.1.4 Training of the Remaining Components 36

5.2 Evaluation 37
5.3 Features Extraction 38
5.4 Video Generation 39
6 TESTING 42
6.1 Model Training 42
6.2 Video Summarizing 43
7 CONCLUSION 45
7.1 Future Work 45
REFERENCES 46

LIST OF FIGURES AND TABLES

Figure 1. Differences between machine learning and software engineering /4/ 12
Figure 2. Types of Machine Learning /5/ 13
Figure 3. Example of a Neural Network /7/ 14
Figure 4. Convolutional layer /10/ 15
Figure 5. Shortcut connection /11/ 17
Figure 6. ResNet and VGG-19 architecture /11/ 17
Figure 7. Example of an RNN /16/ 18
Figure 8. Comparing the structure of an RNN cell and an LSTM cell /17/ 19
Figure 9. The architecture of GAN /21/ 20
Figure 10. Workflow of a VAE /25/ 21
Figure 11. Actor-Critic architecture /29/ 22
Figure 12. The hierarchical structure of a video /31/ 23
Figure 13. Audio wave sampling representation /32/ 24
Figure 14. Benchmarking the performance of different libraries /39/ 28
Figure 15. Application use cases diagram 31
Figure 16. The sequence diagram of the training process 32
Figure 17. The first three steps of the incremental training procedure. Dark-colored boxes denote the parts updated in each step. /41/ 32
Figure 18. The fourth step of the incremental training procedure. Dark-colored boxes denote the parts updated in this step. /41/ 32
Figure 19. The sequence diagram of the summarizing process 33
Figure 20. Visualization of a summarized video 43

Table 1. Application requirement specifications, with 1 indicating highest and 3 indicating lowest priority. 30
Table 2. The evaluation results of the training process (higher is better). 43
Table 3. The video summarizing process evaluation results, with the result lengths being approximately 15% of the original lengths. 44

LIST OF CODE SNIPPETS

Code Snippet 1. Implementation of Knapsack problem in Python /33/ 25

Code Snippet 2. Example of Decord functionality... 28

Code Snippet 3. Example of Pydub functionality 29

Code Snippet 4. Updating the Encoder 34

Code Snippet 5. Updating the Decoder.. 35

Code Snippet 6. Updating Linear Compressor and Discriminator....................... 36

Code Snippet 7. Updating the remaining components of the architecture......... 37

Code Snippet 8. Implementation of evaluation 38

Code Snippet 9. Implementation of feature extraction 39

Code Snippet 10. Implementation of summaries generation............................ 40

Code Snippet 11. Implementation of videos generation 41

Code Snippet 12. Implementation of F-score calculation 42

LIST OF ABBREVIATION

AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
GAN	General Adversarial Network
LSTM	Long Short-term Memory
VAE	Variational Autoencoder
SOTA	State-of-the-art

1 INTRODUCTION

1.1 Background and Motivation

In recent years, video has become a highly significant form of visual data, and the amount of video content uploaded to various online platforms has increased dramatically in recent years. According to Statista /1/, in 2021, YouTube's user base in the world amounts to approximately 2,240.03 million users and might be raised to reach 2,854.14 million users by 2025. In this regard, efficient ways of handling video have become increasingly important.

The explosion of short video platforms, such as TikTok, Instagram, and Facebook has changed our video-consuming behaviour. People tend to consume short content rather than sitting down and watching a 10-minute video without skipping any second. To solve this problem, a video summarizing system is needed, to compress a long video into a much shorter version, but still able preserve the important contents of the original one. Video summarization could be used for summarizing video games, lectures, movies, and so many more.

There are two main ways of summarizing a video right now. One is time-lapse, which is just increases the speed of a video to 5 or even 10 times faster than the original one. Another method is key-shot based, which means dividing a video into multiple segments, then deciding whether a shot is important or not, keeping the important shots only in the end. However, making this kind of decision is a time-consuming and tedious task, and opinions can vary among different people. As a trend in the world nowadays, Artificial Intelligence can be used to maximize work, and it was perfectly suitable for handling this task. /2/

1.2 Objectives

The primary goal of the project was to do research in the field of video summarization and develop a tool that can assist people in improving video summarization workflow by significantly reducing processing time and simplifying video summarizing work.

The question in the research aspect was to select a solution that can provide both high and reliable performance while remaining usable in production. After producing the best solution, the challenge was to develop a tool that can do the summarizing work efficiently, since the original research focused on improving the performance of the method but not on its usability for ordinary users. With the application developed in this project, the user could summarize a video just by inputting the original video file and getting a summarized version of it in terms of seconds.

2 THEORETICAL REVIEW

2.1 Artificial Intelligence

According to IBM, Artificial Intelligence "is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable." /3/ AI is built for maximizing the work of humans, and it can be used for applications such as:

- **Speech recognition** is known as automatic speech recognition (ASR) or speech-to-text (STT). This is an aspect of Natural Language Processing (NLP), which is responsible for processing human speech into a written form like text. There are many systems applying this technology such as voice typing systems, or virtual assistants such as Siri from Apple, Google Assistant from Google, and Alexa from Amazon.
- **Text analyzation** is also another aspect of NLP, which is capable of understanding documents given by humans, then trying to understand that and doing some specified tasks such as sentiment analysis, giving feedback to interact with or used in frequently answered questions (FAQs) system, recommendation system.
- **Computer vision** technology gives the machines the ability to derive meaningful information from digital images, video, or any other kind of visual input, then act based on the given information. Computer vision has some applications such as face recognition, autonomous driving, photo tagging, and the medical field.
- **Recommendation system** is based on the behavior of users in the past, AI algorithms can help to discover data trends that can be used for giving some recommendations to users, such as what we can get from online shopping platforms or search engines.

Both Machine Learning (ML) and Deep Learning (DL) are subfields of AI, and DL is a subfield of ML.

2.2 Machine Learning

Machine Learning is a branch of AI and Computer Science, which focuses on the use of data and algorithms for recreating the way human learns. ML models are built based on data and algorithms as explained in Figure 1, then their accuracy gradually improved by derivatives. The more data is fed into the training model, the better and more accurate the final model would be.

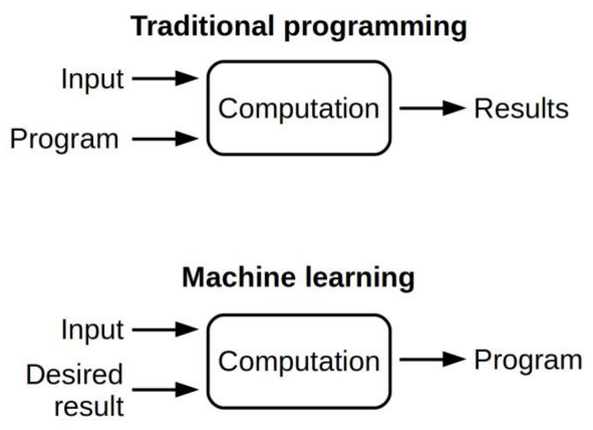


Figure 1. Differences between machine learning and software engineering /4/

There are three main types of ML, as depicted in Figure 2:

- **Supervised Learning**: the model is trained based on a labeled dataset, for serving regression and classification problems, such as predicting if an advertisement is suitable for a person or not, classifying cat and dog pictures. Some algorithms are using this approach like linear regression, random forest, and support vector machines (SVMs).
- **Unsupervised Learning**: the model is trained on neither classified nor labeled datasets, and the model must find the data pattern and optimal solutions on its own. Unsupervised learning is often used in clustering and

abnormal detection problems, abnormal transaction detection in banking as an example. Some famous algorithms built on this method are principal component analysis (PCA), k-means clustering, probabilistic methods. This project is also built based on this method.

- **Reinforcement Learning**: this allows the model to learn in an interactive environment by trial and error using feedback based on its own experiences, or shortly based on reward and penalty.

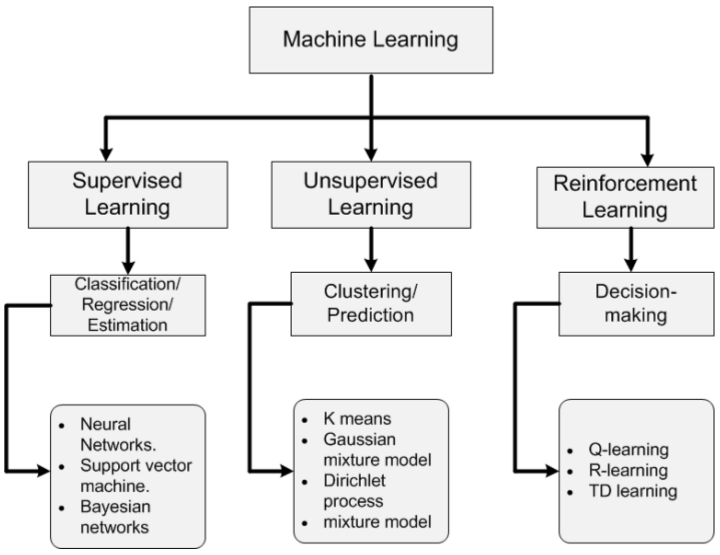


Figure 2. Types of Machine Learning /5/

There is also another method called semi-supervised learning, which is the procedure of using supervised learning to train based on a small amount of labeled data, then use the trained model for labeling the unlabeled data, feed them to train another supervised data.

In the development of ML, a new concept has been developed, named Neural Networks, inspired by our understanding of the biology of our brains, and related to Deep Learning.

2.3 Neural Networks and Deep Learning

According to IBM /6/, a neural network like in Figure 3 replicates the human brain through a set of algorithms. At the basic level, a neural network is created from four things: input, weight, bias, and output. A neural network usually has multiple hidden layers between the input and output, and each hidden layer has its activation function, potentially passing information from the previous one to the next one, like $ReLU$, $Sigmoid$, or $Tanh$.

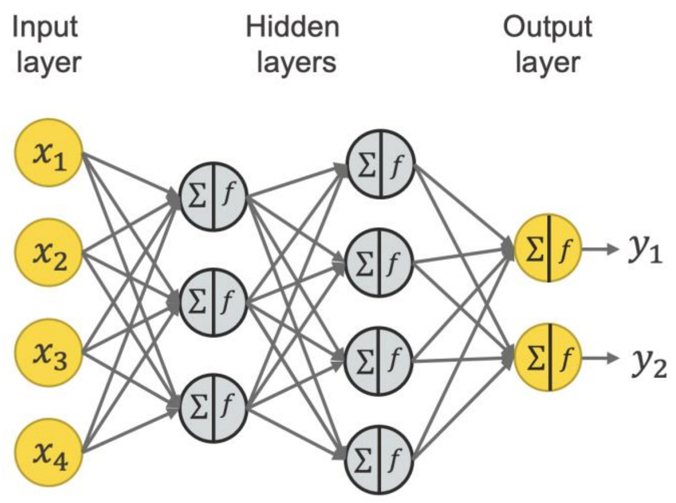


Figure 3. Example of a Neural Network /7/

The main difference between a neural network and a linear function is that a neural network has multiple different neurons, that can give flexibility to the net by changing the weight of a single neuron without affecting others.

Deep Learning (DL) are neural networks with multiple hidden layers /8/. A DL model is trained through the backpropagation, which is moving in the opposite direction from the output to input, instead of flowing from the input to the output. This procedure allows us to make some calculations then give feedback to different neurons, training model through derivatives.

With the development and explosion of DL, there are a lot of different types of neural networks invented, and significantly enhanced the performance of AI, such as Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM), and many more.

2.4 Convolutional Neural Network

Convolutional Neural Network (CNN) /9/ is one of the leading architectures that have been used in a wide variety of practical applications, especially in Computer Vision like Object Classification, Object Detection.

CNN uses a convolutional layer, as explained in Figure 4, which is the core building block of a CNN, for learning the feature representations of the input data. The data can be in the 1D format such as time-series or 2D format such as image data. In each convolutional layer, there is a filter that slides through the input data, creating a feature map for feeding to the non-linear activation functions.

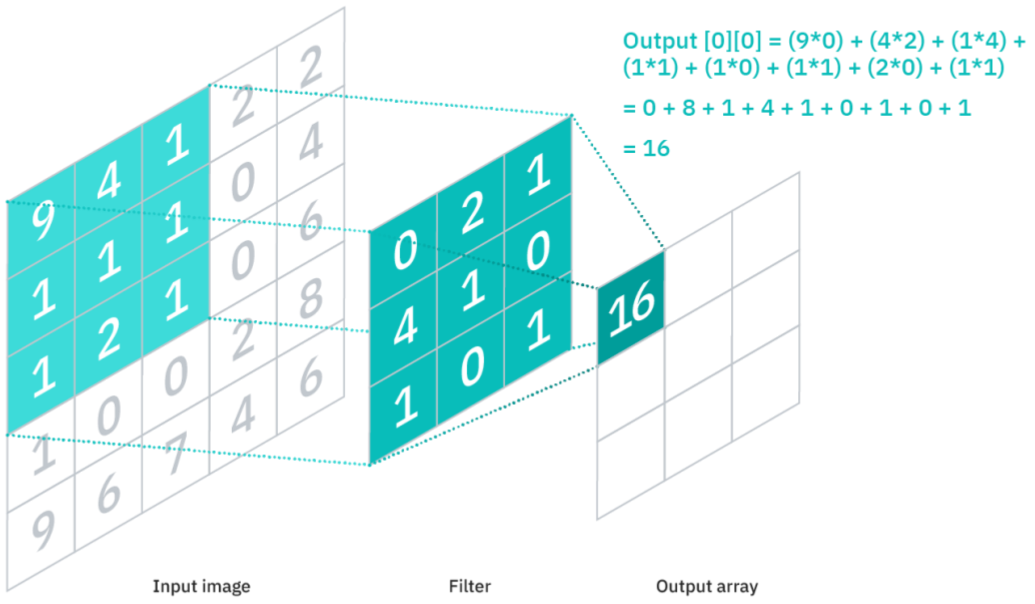

Output [0][0] = (9*0) + (4*2) + (1*4) + (1*1) + (1*0) + (1*1) + (2*0) + (1*1)
= 0 + 8 + 1 + 4 + 1 + 0 + 1 + 0 + 1
= 16
Input image
Filter
Output array

Figure 4. Convolutional layer /10/

Following the convolutional layer, a pooling layer is often used for reducing the number of parameters in the input. Similar to the convolutional layer, there is also a filter that slides through the feature map, but the difference is that there is no learning weight in this filter. According to IBM /10/, normally there are two main types of pooling:

- **Max pooling**: As the filter moves across the input, it selects the pixel with the maximum value to send to the output array. As an aside, this approach tends to be used more often compared to average pooling.
- **Average pooling**: As the filter moves across the input, it calculates the average value within the receptive field to send to the output array.

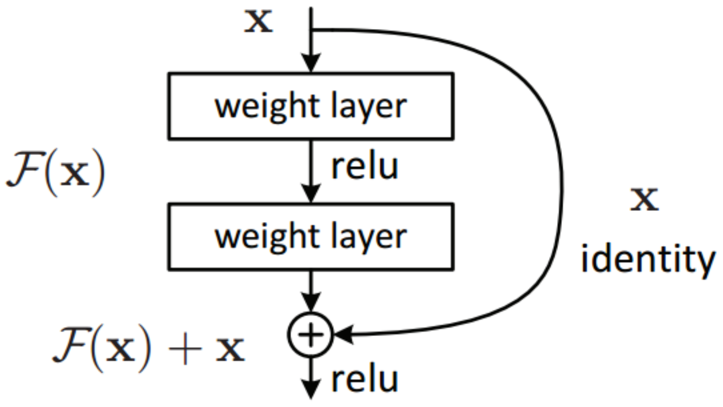
Finally, fully connected layers are often used for high-level specified tasks such as classification, based on the feature extracted from previous layers.

Up until now, there are quite a lot of different variant CNN architectures that have been developed and in use like ResNet /11/, GoogLeNet /12/, VGGNet /13/, and many more.

2.5 ResNet

The rise of CNN has led to a series of breakthrough innovations in image classification, with deeper architectures introduced such as LeNet /12/, VGGNet /13/, and AlexNet /14/. A major pattern observed overall is that networks are designed to be deeper and deeper, to improve the performance of classification tasks. However, deep CNN with too many layers often faces the problem of vanishing/exploding gradient, which affects the convergence of the network and makes the training procedure becomes more challenging.

Announced in December 2015, ResNet /11/ introduces the “shortcut connections” concept, which can solve the vanishing gradient problem and allow researchers to build deeper architectures. As described in Figure 5, shortcut connections are connections that skip one or more layers of a network, mapping stacked layers fit a residual mapping $\mathcal{F}(x)$ instead of directly fitting a desired underlying mapping $\mathcal{H}(x)$. Hence, the output is $\mathcal{H}(x) = \mathcal{F}(x) + x$, and the weight layers are to learn a kind of residual mapping $\mathcal{F}(x) = \mathcal{H}(x) + x$. Even if there is a vanishing gradient for the weight layers, we always still have the identity x to transfer back to earlier layers.



x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 5. Shortcut connection /11/

The overall architecture of ResNet-34 is described in Figure 6, with the comparison of the VGG-19 network. As claimed by authors, ResNet-34 has fewer filters and lower complexity than VGG-19, but gives much better performance, with the vanishing gradient solved. Furthermore, there are 50/101/152 layers variants of ResNet, following the same concept "deeper is better".

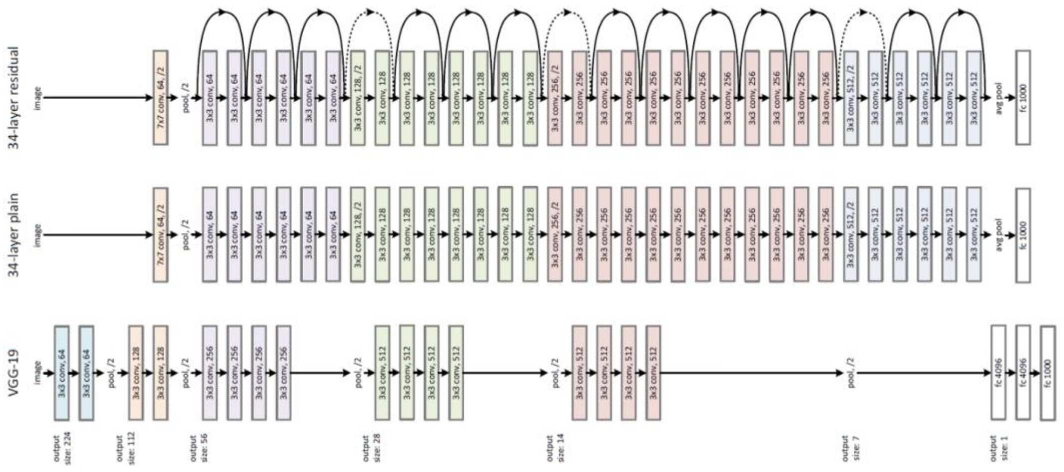


Figure 6. ResNet and VGG-19 architecture /11/

Later, ResNet has also been widely used for feature extraction in other architecture, like Faster-RCNN for object detection /15/, and many more.

2.6 Long Short-Term Memory

LSTM is a special type of Recurrent Neural Network (RNN), which is focused on processing sequential data or time-series data and showing an efficient performance when learning long-term dependencies. /16/

Unlike any other kind of feed-forward neural network, for every element of a sequence, RNN has a special memory to remember computed information of all previous elements. For example, given a sequence of input $(x_1, x_2, \dots x_T)$ in Figure 7, RNN will compute for a total of T timesteps, with each timestep t taking x_t as input, and produce y_t as output. There is a thing called hidden state h_t, which is responsible for remembering all the information of previous elements in the sequence. Hidden state h_t is calculated based on previous hidden state h_{t-1} and input x_t, then output y_t is determined by h_t.

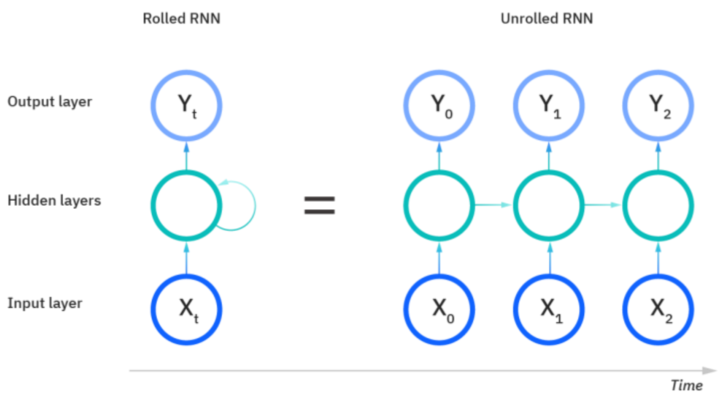


Figure 7. Example of an RNN /16/

With LSTM /18/, each time step is processed through four stages, with two different kinds of state: “hidden state” and “cell state”. In time step t, at the first stage, the LSTM cell will try to decide which information should be thrown away or not. The current input x_t and previous hidden state h_{t-1} will be used for a function called “forget gate” f_t. Values come out between 0 and 1, the closer to 0 means to forget, and the closer to 1 means to keep. Moreover, we have the “input gate” i_t, which is responsible for deciding what information is relevant to add from the current step. We pass the previous hidden state h_{t-1} and current input x_t into a sigmoid σ function, to decide which value should be updated by transforming the value to be between 0 and 1, also regulate the network by passing the previous hidden state h_{t-1} and current input x_t into a $tanh$ function to squish value into -1 and 1 range. After that, the output of σ and $tanh$ function will be multiplied together, deciding what information will be kept from the $tanh$ output, stored by i_t.

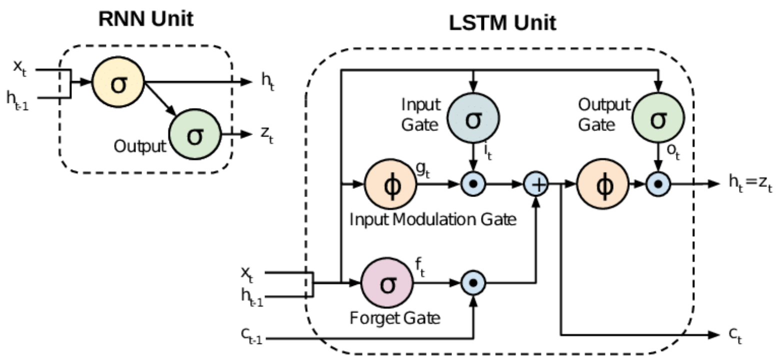


Figure 8. Comparing the structure of an RNN cell and an LSTM cell /17/

After having f_t and i_t calculated, we should now have enough information to calculate the cell state c_t. Previous cell state c_{t-1} gets pointwise multiplied with forget gate f_t, then get a pointwise addition with i_t, producing new cell state c_t. Lastly, we have the output gate o_t, deciding what the next hidden state h_{t+1} should be. o_t is calculated by passing h_{t-1} and x_t into a sigmoid function, then deciding what h_{t+1} should be by multiplying o_t with the result of passing c_t through a $tanh$ function.

For increasing the performance of the LSTM network, a Bidirectional LSTM (BiLSTM) is also used, consisting of two LSTMs: one taking the input in the forward direction, and the other in the backward direction. This effectively increases the amount of information available to the network, improving the context available to the algorithm. /19/ Aside from the normal LSTM, there is also a simple version of it, called Gated Recurrent Unit (GRU), which combines the forget gate and input gate, as well as merges the cell state and hidden state, is also being used widely.

2.7 Generative Adversarial Network

In June 2014, Ian Goodfellow and his colleagues /20/ introduced a class of Machine Learning framework, called Generative Adversarial Network, in which two models are simultaneously trained: a generative network G learns for creating new data that captures the training data distribution, and the discriminative network D learns for discriminating the realistic of generated data. This framework corresponds to a minimax two-player game when G tries to maximize the probability of D making a mistake, while D tries to maximize the errors of G. The training procedure ends when D cannot discriminate the differences between the training data distribution and the distribution of results generated from G.

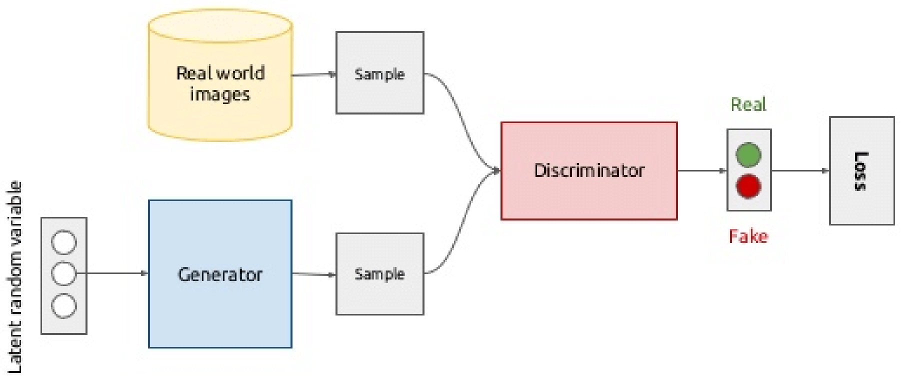


Figure 9. The architecture of GAN /21/

GANs can be used in many different aspects, such as image enhancements, art generating, AR, and VR for reconstructing 3d models. In multimedia processing, GANs can also be used for pictures, videos quality enhancement, and frame predicting. In 2017, there was a problem that has been raised now about the concern of using GANs in creating Deepfakes, which can be used “to manipulate media and replace a real person’s image, voice, or both with similar artificial likenesses or voices. Among the possible risks, deepfakes can threaten cybersecurity, political elections, individual and corporate finances, reputations, and more. This malintent and misuse can play out in scams against individuals and companies, including on social media." /22/

2.8 Variational Autoencoder

A Variational Autoencoder /23,24/ is a type of autoencoder, which the architecture composes of both an encoder and a decoder, trained to minimize the reconstruction error between the encoded-decoded data and the initial data. The training of VAE is regularized to avoid overfitting and ensure that the latent space of the input data has good properties for enabling the generative process, by encoding it as a distribution over latent space instead of encoding as a single point.

As described in Figure 10, the training process of VAE is divided into four steps:

- First, the input is encoded as a distribution over the latent space
- Second, a point from the latent space is sampled from that distribution
- Third, the sampled point is then decoded, and the reconstruction error can be computed
- Finally, the reconstruction error is backpropagated through the network

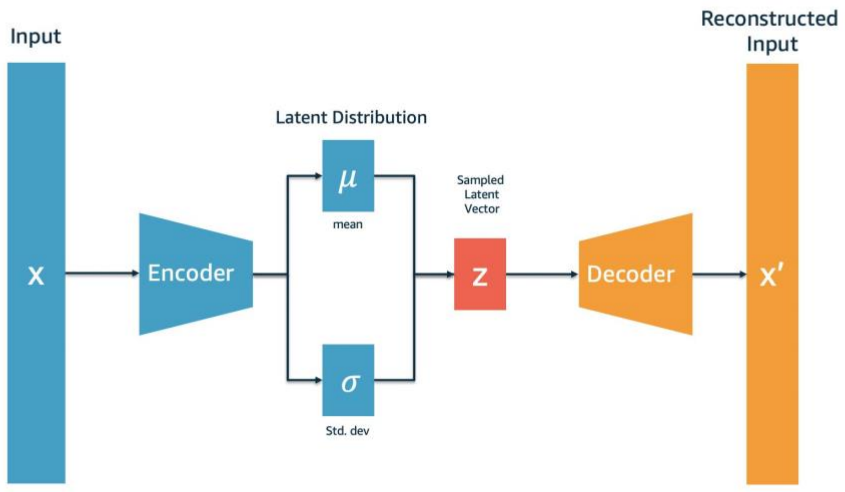


Figure 10. Workflow of a VAE /25/

There are different studies on the combination of VAE and GAN /26,27/ since the Decoder module of VAE and the Generator module of GAN share some similarities in the generation tasks.

2.9 Actor-Critic Model

Actor-Critic is a method in Reinforcement Learning, which combines the advantages of actor-only and critic-only methods, where the words actor and critic are synonyms for the policy and value function, respectively. The principal idea is to split the model into two: one for computing an action based on a state and another one to produce the Q values of the action. The actor is responsible for deciding which action should be taken, and the critic will inform the actor how good was the action and how should it adjust. Those two models together participate in a game where they both get better in their roles. The result is that the overall architecture will learn to play the game more efficiently than the two methods separately. /28/

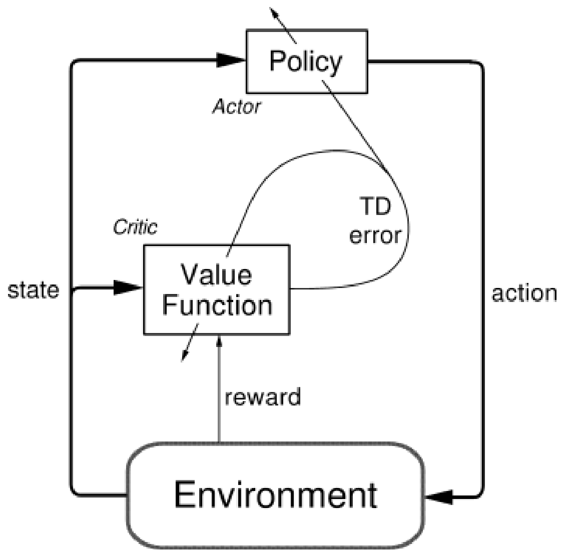


Figure 11. Actor-Critic architecture /29/

Actor-Critic and GANs also share some similarities, such as "one model has access to information about errors from the environment (the discriminator in GANs and the critic in AC), while the other model must be updated based only on gradient information from the first model", so that there are many studies on the combination of two. /30/

2.10 Video Structure

A video is made up of a series of various frames, also known as images, that are displayed at a constant rate, such as 24 frames per second (fps), 30fps, and 60fps, as seen in Figure 12. A shot is made up of consecutive frames with similar features, different shots are put together to make a scene, and scenes are joined together to form a video.

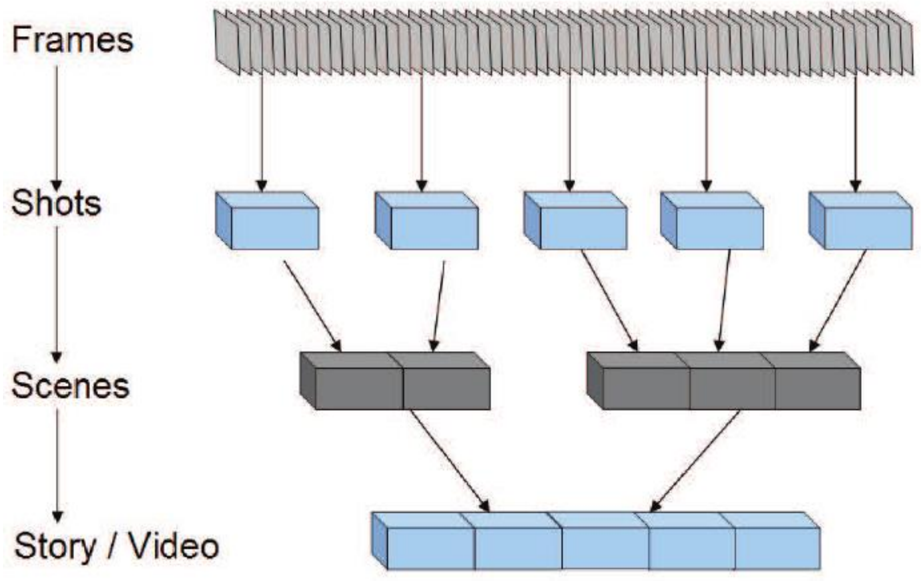


Figure 12. The hierarchical structure of a video /31/

Most videos now include audio tracks that are played in parallel with visuals to provide additional information. As shown in Figure 13, audios are represented in digital form with a variety of different sample points at a constant rate, such as 44,1kHz. Speakers can play them constantly, and accurately represent the audio information in the video using these values.

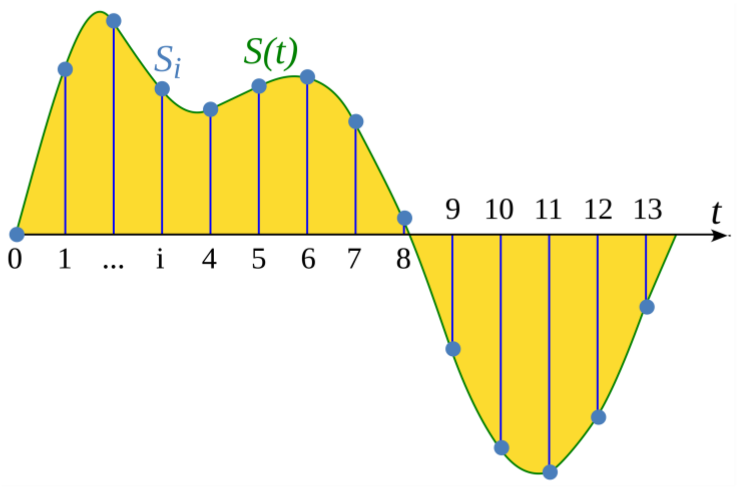


Figure 13. Audio wave sampling representation /32/

2.11 Knapsack Algorithm

The knapsack problem is one of the combinatorial optimization problems: given a set of n items, and two integer arrays $val[0..n-1]$ and $wt[0..n-1]$ represent values and weights associated with n items respectively; also, given a knapsack with the capacity of W, determine the maximum subset of $val[]$ such that the total weight is less than or equal to the capacity W. It has been studied for more than a century since 1897, referring to the commonplace problem of packing the most valuable or useful items without overloading the luggage /33/. In the real-world, knapsack problems are applied to find the least wasteful way to cut raw materials, selection of investment and portfolios, selection of assets for asset-backed securitization, and many more.

The knapsack problem can be solved by dynamic programming, with the time and space complexity of $O(n * W)$. We will use a two-dimensional array $K[][]$ for remembering the states of calculation, with weights from 1 to W as the columns and items from 1 to n as the rows; the state $K[i][j]$ will denote maximum value of $j - th$ weight considering values of items from 1 to $i - th$. So, if we consider $wt[i]$, we can fill it in all columns that the weight values are greater than $wt[i]$. At this stage, there are two possibilities: fill $wt[i]$ into the given column or not, which means we must take the maximum value of these possibilities:

- If we do not fill the $wt[i]$ in the $j\text{– }th$ column, $K[i][j]$ will be the same as $K[i-1][j]$
- If we fill $wt[i]$, $K[i][j]$ will be equals to $val[i]+K[i-1][j-[wt[i]]$

The optimal solution will be the value at $K[n][W]$ state. /33/ Code Snippet 1 below shows how the solution is implemented in Python.

```
def knapsack(W, wt, val, n):
    K = [[0 for x in range(W+1)] for x in range(n+1)]

    # Build table K[][] in bottom up manner
    for i in range(n+1):
        for w in range(W+1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1]+K[i-1][w-wt[i-1]],
                    K[i-1][w])
            else:
                K[i][w] = K[i-1][w]

    # Trace the selected items
    selected = []
    w = W
    for i in range(n, 0, -1):
        if K[i][w] != K[i-1][w]:
            selected.insert(0, i-1)
            w -= wt[i-1]

    return selected
```

Code Snippet 1. Implementation of Knapsack problem in Python /33/

3 TOOLS AND TECHNOLOGIES IN USE

In this chapter, the tools and technologies that have been used for developing the application are discussed. Since the application is deep learning-based, PyTorch (Python) was mainly used for the training and evaluating process. Since we are dealing with multimedia issues, we also use OpenCV, Decord, and Pydub.

3.1 Python

Python is a simple, interpreted, object-oriented, high-level programming language with dynamic semantics /34/. Clear and simple syntax makes it widely used and gains more and more applications internationally such as web development, data visualization, data analytics, and especially AI and machine learning. With most of the features of an object-oriented language for full object-oriented programming and cross-platform for a variety of operating systems including Windows, macOS, and Linux, there are a lot of different frameworks developed for AI and data science such as PyTorch, Tensorflow, Keras, mxnet, NumPy, Pandas, and many more.

3.2 PyTorch

Released to open source in 2017, PyTorch /35/ is a machine learning framework developed by Facebook’s AI Research lab. According to Paszke et al. /36/, “it provides an imperative and Pythonic programming style that supports code as a model, makes debugging easy and is consistent with other popular scientific computing libraries, while remaining efficient and supporting hardware accelerators such as GPUs.”

Not only being integrated with popular libraries such as NumPy, SciPy and supports CPU, GPU, but also PyTorch supports parallel processing, as well as distributed training, and having excellent documentation with a large community that is active and supportive, makes it a great option for implementing Computer Vision (CV), Natural Language Processing (NLP) and many other tasks. /37/

3.3 OpenCV

OpenCV is an open-source library for machine learning and computer vision, mainly aimed at real-time processing, widely adopted with an estimated number of exceeding 14 million downloads. With more than 2500 optimized algorithms, which include a comprehensive set of both classic and state-of-the-art (SOTA) computer vision and machine learning algorithms, it can be used to detect and recognize faces, identify objects, track moving objects, and many more. Since this library is primarily focused on real-time performance, it was optimized by C/C++, was designed for computational efficiency, and can take advantage of multi-core processing capabilities. Furthermore, it can take advantage of the hardware acceleration of the underlying heterogeneous compute platform by supporting OpenCL and CUDA frameworks. It has C++, Python, Java, and MATLAB interfaces and supports Windows, Linux, Android, and macOS, so there are large companies such as Google, Yahoo, Microsoft, Intel, and IBM making extensive use of this library, as well as start-ups such as Applied Minds, VideoSurf, and Zeitera. /38/

3.4 Decord

Announced in CVPR 2020, Decord is a library that provides convenient video slicing methods based on a thin wrapper on top of hardware-accelerated video decoders, for example, FFMPEG/LibAV, Nvidia Codecs, and Intel Codecs /39/. The random-access pattern of this library is drastically improved comparing to the performance of any other libraries, particularly OpenCV. Figure 14 shows the performance comparison between Decord and OpenCV, with up to 2.38 times faster at sequential read and up to 14.48 times faster at accurate random seek. With dramatically improved efficiency, this library simplifies video processing tasks, allowing Python to become a much more efficient programming language.

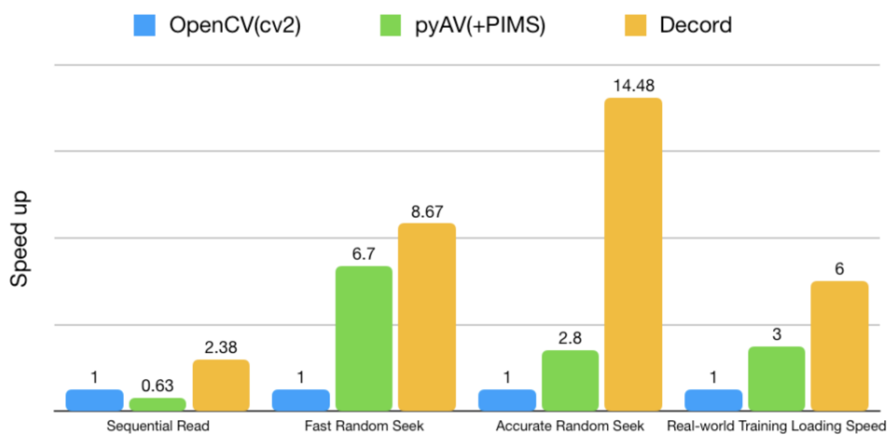


Figure 14. Benchmarking the performance of different libraries /39/

Decord is also capable of decoding audio from both video and audio files. One can slice video and audio together to get a synchronized result, hence providing a one-stop solution for both video and audio decoding. Code Snippet 2 shows an example of Decord functionality.

```
from decord import VideoReader

vr = VideoReader('examples/test.mp4')

print('Video frames:', len(vr))
# Output: 7414

print('Frame shape:', vr[0].shape)
# Output: (720, 1280, 3)

print('Batch of frames shape:', vr[0:10].shape)
# Output: (10, 720, 1280, 3)

print('Key frames list:', vr.get_key_indices())
# Output: [0, 44, 85, 125, 161, 221, 260, 289]

print('FPS:', vr.get_avg_fps())
# Output: 25.0
```

Code Snippet 2. Example of Decord functionality

3.5 Pydub

Pydub /40/ is a Python library for handling audio files. Similar to Decord, Pydub allows slicing and locating audio files with different metrics such as timestamps or indexes easier. Moreover, it also offers the ability of audio editing for example, volume modification, audio concatenating, adding crossfade, and many more. It supports a wide range of codecs, from audio formats such as WAV, MP3, and AAC to video formats such as MP4, FLV, and WMV. Code Snippet 3 below shows a functionality example of Pydub.

```
from pydub import AudioSegment
from pydub.playback import play

ar = AudioSegment.from_file('examples/test.mp4', 'mp4')

# Pydub does things in milliseconds
first_10_seconds = ar[:10*1000]
last_5_seconds = ar[-5000:]

# Boost volume by 10dB
beginning = first_10_seconds + 10
# Reduce volume by 5dB
end = last_5_seconds - 5

# Concatenate audio
combined = beginning + end
play(combined)

print('Length of audio:', combined.duration_seconds)
# Output: 15.0

# Save audio
combined.export('test.mp3', format='mp3')
```

Code Snippet 3. Example of Pydub functionality

4 APPLICATION DESCRIPTION

In this chapter, the structure and requirement specifications of the application will be discussed.

This project used state-of-the-art research in the unsupervised video summarization field, which is AC SUM GAN /41/ as the logical part for processing. The tool was focused on improving the workflow and simplifying the work of video summarization just by inputting the original video file and getting a summarized version of it. The requirements of this project could be categorized into three distinct levels of importance: must-have, should have, and nice to have, with the order of priorities are #1, #2, and #3 respectively. Table 1 fully describes the requirement specifications of this application.

Table 1. Application requirement specifications, with 1 indicating highest and 3 indicating lowest priority.

Reference	Description	Priority
F1	The application can receive an input video, then produce a summarized video	1
F2	The application could easily be used with the command-line interface	1
F3	Output video should include audio, exactly matching to the scenes selected by the application	1
F4	Performance of the solution should be reasonably close to the original solution from the paper	2
F5	Input pre-processing time should be minimized so that the overall process can be reduced	2
F6	The application should allow users to manipulate the length of the output video	3

Based on the requirement specifications of the summarizing application, there were two main use cases that the application can serve: train the main processing model and summarize videos, matching the use of developers and normal users, respectively. Figure 15 depicts the use cases diagram of the application.

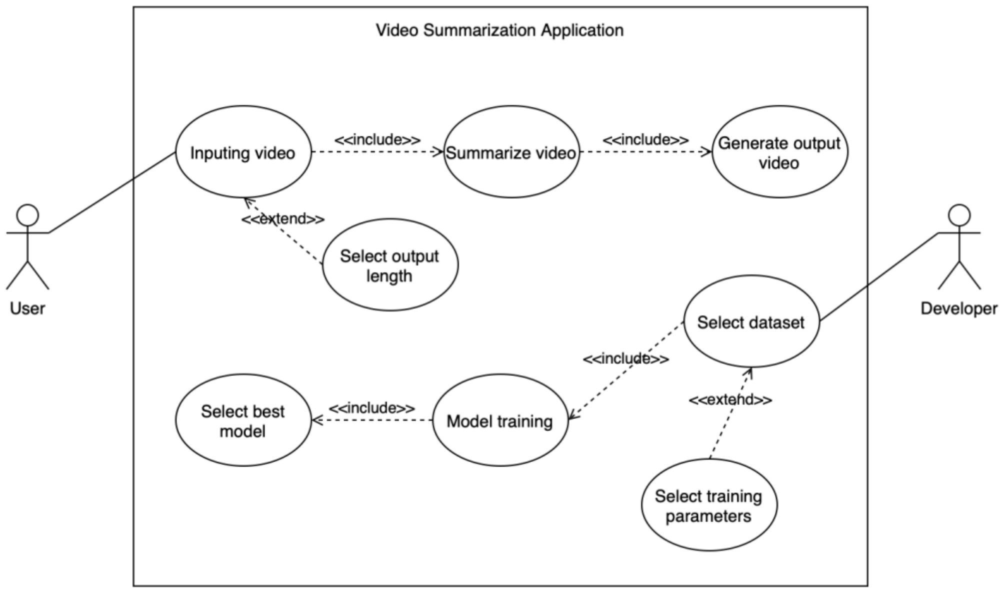


Figure 15. Application use cases diagram

According to Apostolidis et al. /41/, a training epoch was divided into four incremental steps, and at each step, different components were being trained (e.g., Linear Encoder, Encoder, Decoder). Figures 17 and 18 show how components were being trained through each step, with the first three steps of the procedure depicted in Figure 17, and the step 4 of the procedure depicted in Figure 18. This procedure was looped through 100 epochs, then the best model would be chosen for the testing part. Figure 16 shows the sequence diagram of the whole training process.

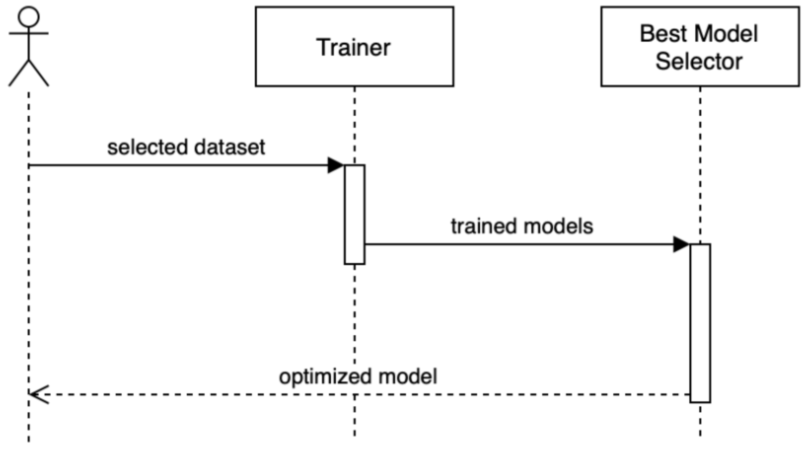


Figure 16. The sequence diagram of the training process

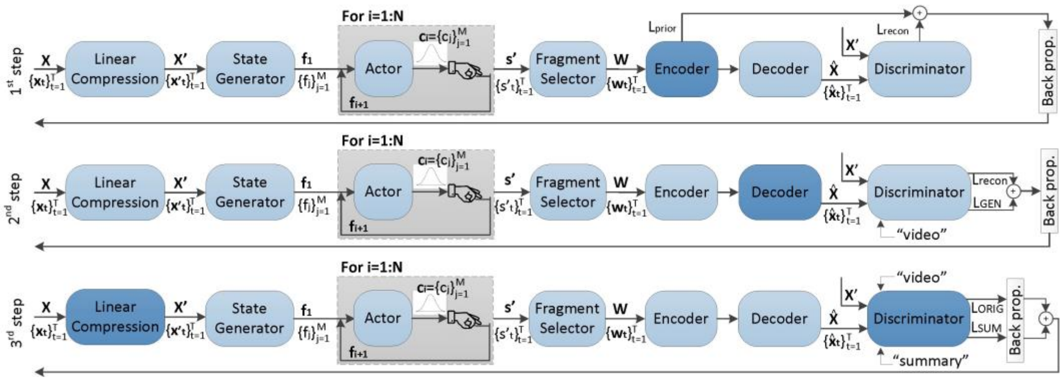


Figure 17. The first three steps of the incremental training procedure. Dark-colored boxes denote the parts updated in each step. /41/

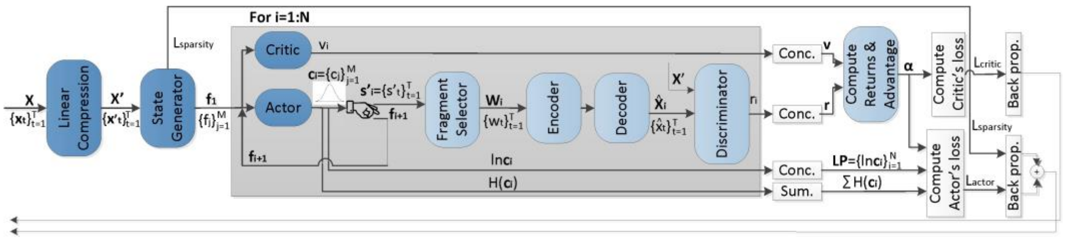


Figure 18. The fourth step of the incremental training procedure. Dark-colored boxes denote the parts updated in this step. /41/

In the video summarizing part, the input video features would be extracted first. Firstly, we must recognize the keyframes in the input video, then divide them into shots for evaluation. for examination. The KTS algorithm /42/ could be used to perform this technique, however, it usually takes a long time since it would look over every individual frame of the video before deciding on the keyframes. Fortunately, by using Decord here, we could greatly minimize the processing time, so that we could obtain the result in just a few seconds.

After obtaining keyframes and partitioning the original video into shots, we took the representative frames of each shot and feed them into a pre-trained ResNet for extracting features. Then, all the extracted features were evaluated using the pre-trained model in the training part, scoring the importance of shots, then using the Knapsack algorithm for deciding which shots were kept or not, calculating the most valuable result in a limited period decided in the beginning. Finally, chosen shots were then combined, forming the final output video. Figure 19 shows how different modules of the process interact with each other.

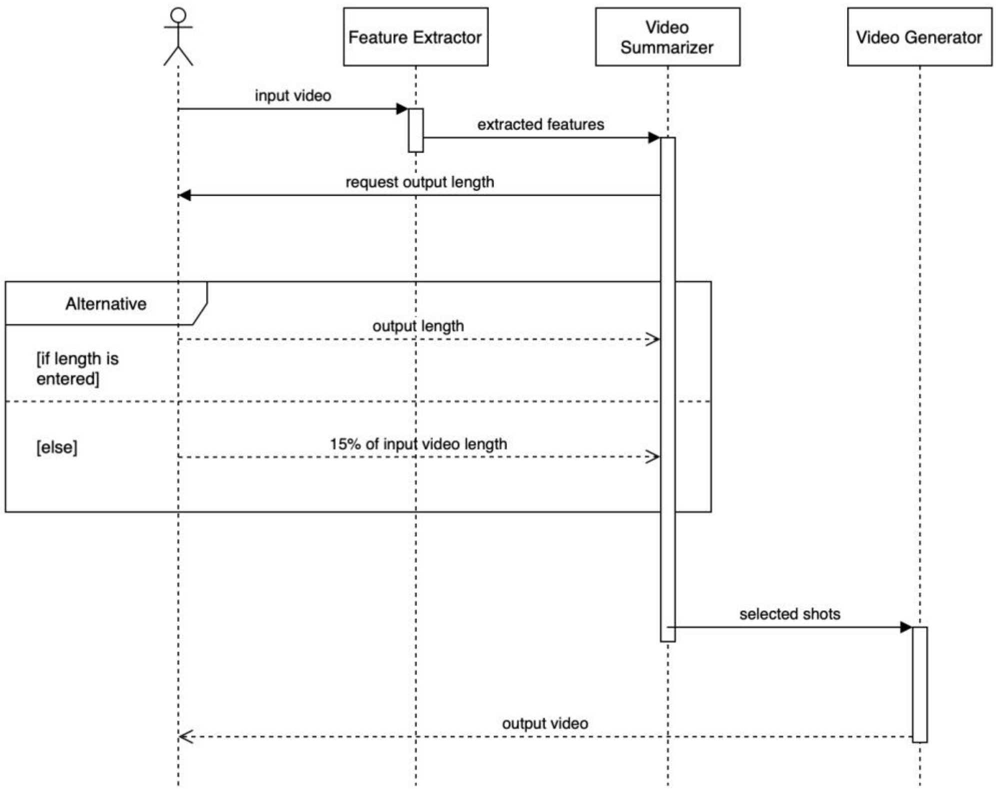


Figure 19. The sequence diagram of the summarizing process

5 IMPLEMENTATION

In this section, the process of implementing the application is discussed.

5.1 Training

As described in Figures 17 and 18, the training pipeline of this application was divided into four incremental steps. The Encoder would be trained in the first step, then the Decoder would be trained in the second step. The third step would be used for training the Discriminator and Linear Compression modules. Finally, in the fourth step, the State Generator, Critic, and Actor modules would be trained, and the Linear Compression module was once again being learned. The whole training process was looped through 100 epochs, for producing the best result.

5.1.1 Training Encoder

```
#---------- train eLSTM ----------#
e_optimizer.zero_grad()
for video in range(config.batch_size):
    image_features, action_fragments = next(iterator)
    image_features = image_features.view(-1, config.input_size)
    action_fragments = action_fragments.squeeze(0)
    image_features_ = Variable(image_features)
    seq_len = image_features_.shape[0]
    original_features = linear_compress(image_features_
        .detach()).unsqueeze(1)
    weighted_features, _ = AC(original_features,
        seq_len, action_fragments)
    h_mu, h_log_variance, generated_features = summarizer
        .vae(weighted_features)
    h_origin, original_prob = discriminator(original_features)
    h_summary, summary_prob = discriminator(generated_features)
    recon_loss = reconstruction_loss(h_origin, h_summary)
    prior_loss = prior_loss(h_mu, h_log_variance)
    e_loss = recon_loss + prior_loss
    e_loss = e_loss/config.batch_size
    e_loss.backward()
# update e_lstm parameters every 'batch_size' iteration
torch.nn.utils.clip_grad_norm_(summarizer.vae.e_lstm
    .parameters(), config.clip)
e_optimizer.step()
```

Code Snippet 4. Updating the Encoder

Depicted in Figure 17, in the first step, the algorithm made a forward pass through all the components of the network (Linear Compression, State Generator, Actor, Fragment Selector, Encoder, Decoder, and Discriminator) for computing losses. The prior loss L_{prior} was calculated by using the Encoder module, and the reconstruction loss L_{recon} was calculated by using the Discriminator. After L_{prior} and L_{recon} was calculated, the Encoder was updated with a backward. The entire process of step 1 was implemented in Code Snippet 4.

5.1.2 Training Decoder

Similar to step 1, in step 2, the input was forwarded through the partially updated network as in Figure 17, computing the reconstruction loss L_{recon} and the generator loss L_{GEN} using the Discriminator. L_{recon} and L_{GEN} was then summed together, updating the Decoder module. The procedure of this step is described in Code Snippet 5.

```
#---------- train dLSTM ----------#
d_optimizer.zero_grad()
for video in range(config.batch_size):
    image_features = list_image_features[video]
    action_fragments = list_action_fragments[video]
    image_features_ = Variable(image_features)
    seq_len = image_features_.shape[0]
    original_features = linear_compress(image_features_
        .detach()).unsqueeze(1)
    weighted_features, _ = AC(original_features,
        seq_len, action_fragments)
    _, _, generated_features = summarizer.vae(weighted_features)
    h_origin, original_prob = discriminator(original_features)
    h_summary, summary_prob = discriminator(generated_features)
    recon_loss = reconstruction_loss(h_origin, h_summary)
    gen_loss = criterion(summary_prob, original_label)
    orig_features = original_features.squeeze(1)
    gen_features = generated_features.squeeze(1)
    recon_losses = []
    for frame_index in range(seq_len):
        recon_losses.append(reconstruction_loss(orig_features
            [frame_index,:], gen_features[frame_index,:]))
    recon_loss_init = torch.stack(recon_losses).mean()
    d_loss = recon_loss + gen_loss
    d_loss = d_loss/config.batch_size
    d_loss.backward()
# update d_lstm parameters every 'batch_size' iteration
torch.nn.utils.clip_grad_norm_(summarizer.vae.d_lstm
    .parameters(), config.clip)
d_optimizer.step()
```

Code Snippet 5. Updating the Decoder

5.1.3 Training Linear Compression and Discriminator

In step 3, the partially model was again forward passed as shown in Figure 17. The compressed features (which are produced by the Linear Compressor) were then fed into the Discriminator, calculating the original loss L_{ORIG}. The features reconstructed by the Decoder were also fed into the Discriminator, calculating the generator loss L_{GEN}. The gradients computed from the two losses, after two individual backward passes, were then used for updating the Discriminator and the Linear Compressor. Implementation of this step is shown in Code Snippet 6.

```
#---------- train cLSTM ----------#
c_optimizer.zero_grad()
for video in range(config.batch_size):
    image_features = list_image_features[video]
    action_fragments = list_action_fragments[video]
    image_features_ = Variable(image_features)
    seq_len = image_features_.shape[0]
    # Train with original loss
    original_features = linear_compress(image_features_
        .detach()).unsqueeze(1)
    _, original_prob = discriminator(original_features)
    c_original_loss = criterion(original_prob, original_label)
    c_original_loss = c_original_loss/config.batch_size
    c_original_loss.backward()
    # Train with summary loss
    weighted_features, _ = AC(original_features,
        seq_len, action_fragments)
    _, _, generated_features = summarizer.vae(weighted_features)
    _, summary_prob = discriminator(generated_features)
    c_summary_loss = criterion(summary_prob, summary_label)
    c_summary_loss = c_summary_loss/config.batch_size
    c_summary_loss.backward()
# update c_lstm parameters every 'batch_size' iteration
torch.nn.utils.clip_grad_norm_(list(discriminator.parameters())
    + list(linear_compress.parameters()), config.clip)
c_optimizer.step()
```

Code Snippet 6. Updating Linear Compressor and Discriminator

5.1.4 Training of the Remaining Components

In the last step, the State Generator, the Actor, the Critic, and the Linear Compressor were all updated through an incremental process as shown in Figure 18, with the partially updated model. At the end of the process, actor loss L_{actor}, critic loss L_{critic}, and sparsity loss $L_{sparsity}$ were calculated, making backward steps for the remaining components of the architecture. This process implementation is fully described in Code Snippet 7 below.

```
#---------- train sLSTM, Actor and Critic ----------#
actor_s_optimizer.zero_grad()
critic_optimizer.zero_grad()
for video in range(config.batch_size):
    image_features = list_image_features[video]
    action_fragments = list_action_fragments[video]
    image_features_ = Variable(image_features)
    seq_len = image_features_.shape[0]
    original_features = linear_compress(image_features_
        .detach()).unsqueeze(1)
    next_state, log_probs, values, rewards,
        masks, entropy = AC_s4(original_features,
        seq_len, action_fragments)
    next_state = torch.FloatTensor(next_state)
    next_value = critic(next_state)
    returns = compute_returns(next_value, rewards, masks)
    log_probs = torch.cat(log_probs)
    returns = torch.cat(returns).detach()
    values = torch.cat(values)
    advantage = returns - values
    actor_loss = -((log_probs*advantage.detach()).mean()
        +(config.entropy_coef/config.termination_point)*entropy)
    sparsity_loss = sparsity_loss(scores)
    critic_loss = advantage.pow(2).mean()
    actor_loss = actor_loss/config.batch_size
    sparsity_loss = sparsity_loss/config.batch_size
    critic_loss = critic_loss/config.batch_size
    actor_loss.backward()
    sparsity_loss.backward()
    critic_loss.backward()
# update s_lstm, actor and critic parameters every 'batch_size'
iteration
torch.nn.utils.clip_grad_norm_(list(actor.parameters())
    + list(linear_compress.parameters())
    + list(summarizer.s_lstm.parameters())
    + list(critic.parameters()), config.clip)
actor_s_optimizer.step()
critic_optimizer.step()
```

Code Snippet 7. Updating the remaining components of the architecture

5.2 Evaluation

After the completion of the training process, the successfully trained model was then saved and used to calculate the importance scores of various shots in a video. The input video was fed into the Linear Compressor, which reduces the size of feature vectors from 1024 to 512 before using the State Generator, Actor, and Critic to score the importance of different shots in the original video. The evaluated result was then saved to a JSON file for later uses. Code Snippet 8 below describes the entire process implementation.

```
def evaluate(epoch_i):
    model.eval()
    out_dict = {}
    for image_features, video_name, action_fragments in tqdm(
        test_loader, desc='Evaluate', ncols=80, leave=True):

        image_features = image_features.view(-1,
            config.input_size)
        image_features_ = Variable(image_features)

        original_features = linear_compress(image_features_
            .detach()).unsqueeze(1)
        seq_len = original_features.shape[0]

        with torch.no_grad():
            _, scores = AC(original_features, seq_len,
                action_fragments)

            scores = scores.squeeze(1)
            scores = scores.cpu().numpy().tolist()
            out_dict[video_name] = scores

        score_save_path = config.score_dir
            .joinpath(f'{config.video_type}_{epoch_i}.json')
        if not os.path.isdir(config.score_dir):
            os.makedirs(config.score_dir)
        with open(score_save_path, 'w') as f:
            json.dump(out_dict, f)
        score_save_path.chmod(0o777)
```

Code Snippet 8. Implementation of evaluation

5.3 Features Extraction

Before evaluation, the original video had to be preprocessed first. To begin, the video was divided into different sub-shots by detecting the keyframes (changing points), then we took some representative frames of that shot, passing through ResNet for extracting primary features. These features, as well as other metadata such as frames per second, change points, and sequences, were then saved to a compressed file, and could be evaluated by the trained model then.

The KTS Algorithm /42/ could be used to implement keyframe detection, but it typically takes a long time to process the result (normally hours). Fortunately, the result could be retrieved in seconds by using Decord, significantly reducing processing time. Code Snippet 9 explains how the feature extraction process was carried out.

```
def generate_dataset():
    for video_idx, video_path in enumerate(tqdm(video_list,
        desc='Feature Extract', ncols=80, leave=True)):

        video_name = os.path.basename(video_path)
        # for passing through resnet
        vr = decord.VideoReader(video_path, width=224,
            height=224)
        fps = vr.get_avg_fps()
        n_frames = len(vr)
        frame_list, picks = [], []
        video_feat = None
        change_points, n_frame_per_seg = get_change_points(
            video_path)
        # mid frame of shot representing main features
        for segment in change_points:
            mid = (segment[0] + segment[1])//2
            frame = vr[mid].asnumpy()
            frame_feat = extract_feature(frame)
            picks.append(mid)
            if video_feat is None:
                video_feat = frame_feat
            else:
                video_feat = np.vstack((video_feat, frame_feat))
        h5_file['video_' + video_idx]
            ['features'] = list(video_feat)
        h5_file['video_' + video_idx]
            ['picks'] = np.array(list(picks))
        h5_file['video_' + video_idx]['n_frames'] = n_frames
        h5_file['video_' + video_idx]['fps'] = fps
        h5_file['video_' + video_idx]
            ['change_points'] = change_points
        h5_file['video_' + video_idx]
            ['n_frame_per_seg'] = n_frame_per_seg
        h5_file['video_' + video_idx]['video_name'] = video_name
    h5_file.close()
```

Code Snippet 9. Implementation of feature extraction

5.4 Video Generation

The final result would be generated after the evaluation step. Metadata from the feature extraction step, as well as evaluated scores from the evaluation step, were then retrieved and used in the sub-shot selection process. The Knapsack algorithm was used to complete the sub-shot selection process, with a "capacity of the bag" (length of the output video) inputted. Code Snippet 10 shows how the shots selection process is implemented.

```
def generate_summaries(score_path, metadata_path, duration=-1):
    all_scores = []
    with open(score_path) as f:
        data = json.loads(f.read())
        keys = list(data.keys())
        for video_name in keys:
            scores = np.asarray(data[video_name])
            all_scores.append(scores)

    video_names, all_shot_bound = [], []
    all_nframes, all_positions = [], []
    with h5py.File(metadata_path, 'r') as hdf:
        for video_key in keys:
            video_index = video_key[6:]

            video_name = hdf[video_key + '/video_name'][()]
                .decode()
            sb = np.array(hdf.get('video_' + video_index
                + '/change_points'))
            n_frames = np.array(hdf.get('video_' + video_index
                + '/n_frames'))
            positions = np.array(hdf.get('video_' + video_index
                + '/picks'))

            video_names.append(video_name)
            all_shot_bound.append(sb)
            all_nframes.append(n_frames)
            all_positions.append(positions)
    all_summaries = generate_summary(all_shot_bound, all_scores,
        all_nframes, all_positions, duration)

    return video_names, all_summaries
```

Code Snippet 10. Implementation of summaries generation

Following the completion of the sub-shots selection step, the selected results were combined to make a complete output video, which includes a 5-second intro and outro. Audio segments associated with the selected video shots were also retrieved, concatenated, and saved as a file. After that, the created video and audio were combined to form the final product. The implementation of the video generation process is shown in Code Snippet 11.

```
def generate_videos(video_names, all_summaries):
    for video_name, summary in zip(video_names, all_summaries):
        audio_name = video_name[:-4] + '.mp3'
        video_reader = decord.VideoReader(tmp_path)
        audio_reader = AudioSegment.from_file(tmp_path, 'mp4')
        fps = video_reader.get_avg_fps()
        (frame_height, frame_width, _) = video_reader[0]
            .asnumpy().shape
        # add 5 seconds of video beginning and end into summary
        summary[:int(fps*5)] = 1
        summary[-int(fps*5):] = 1
        frame_ids = list(np.argwhere(summary == 1)
            .reshape(1, -1).squeeze(0))
        vid_writer = cv2.VideoWriter(
            'output_video/' + video_name,
            cv2.VideoWriter_fourcc(*'mp4v'),
            fps, (frame_width, frame_height))
        summarized_audio = None
        for idx in frame_ids:
            # write video to file
            frame = video_reader[idx]
            vid_writer.write(cv2.cvtColor(frame.asnumpy(),
                cv2.COLOR_RGB2BGR))
            au_start, au_end = video_reader
                .get_frame_timestamp(idx)
            # seconds to miliseconds
            au_start = round(au_start*1000)
            au_end = round(au_end*1000)
            if summarized_audio is None:
                summarized_audio = audio_reader[au_start:au_end]
            else:
                summarized_audio +=audio_reader[au_start:au_end]
        # write audio to file
        summarized_audio.export('output_video/' + audio_name,
            format='mp3')
        vid_writer.release()
        # combine video and audio
        input_video = mpe.VideoFileClip(
            'output_video/' + video_name)
        input_audio = mpe.AudioFileClip(
            'output_video/' + audio_name)
        output_video = input_video.set_audio(input_audio)
        output_video.write_videofile(
            'output_video/fin_' + video_name,
            codec='libx264', audio_codec='aac')
```

Code Snippet 11. Implementation of videos generation

6 TESTING

6.1 Model Training

The training procedure was compared to the performance of the AC-SUM-GAN, as published by Apostolidis et al., /41/. TVSum dataset /43/, which is a standardized dataset in the video summarization field, containing 50 videos of various genres (e.g., news, how-to, documentary, vlog, egocentric) and 1,000 annotations of shot-level importance scores obtained via crowdsourcing (20 per video), was used for this benchmark, with five different splits proposed by Apostolidis et al. Each split was trained for 100 epochs, then the best epoch was chosen for comparison based on Reward and Actor loss.

The F-score of selected epochs were calculated as shown in Code Snippet 12, with precision calculated by the overlapped to generated ratio and recall calculated by the overlapped to ground truth ratio. The results of the benchmark can be found in Table 2.

```
def evaluate_summary(predicted_summary, user_summary,
    eval_method):

    max_len = max(len(predicted_summary), user_summary.shape[1])
    S = np.zeros(max_len, dtype=int)
    G = np.zeros(max_len, dtype=int)
    S[:len(predicted_summary)] = predicted_summary
    f_scores = []
    for user in range(user_summary.shape[0]):
        G[:user_summary.shape[1]] = user_summary[user]
        overlapped = S & G
        precision = sum(overlapped)/sum(S)
        recall = sum(overlapped)/sum(G)
        if precision + recall == 0:
            f_scores.append(0)
        else:
            f_scores.append(
                2*precision*recall*100/(precision+recall))
    if eval_method == 'max':
        return max(f_scores)
    else:
        return sum(f_scores)/len(f_scores)
```

Code Snippet 12. Implementation of F-score calculation

Table 2. The evaluation results of the training process (higher is better).

Split	Selected epoch's F-score	The reference's F-score
0	57.3	60.6
1	59.6	
2	57.3	
3	58.5	
4	61.0	

As shown in Table 2, the training process yielded results that were close to the original implementation, indicating that the application was already in a good shape and capable of producing a well-summarized video.

6.2 Video Summarizing

For testing the video summarizing process, 16 videos taken from YouTube were used, with the predefined output length was 15% of the input. The results of summarizing process are shown in Table 3. As we can see, the result lengths were approximately equal to 15% of the original length, matching the predefined configuration. Figure 20 shows the visualization of a summarized video.

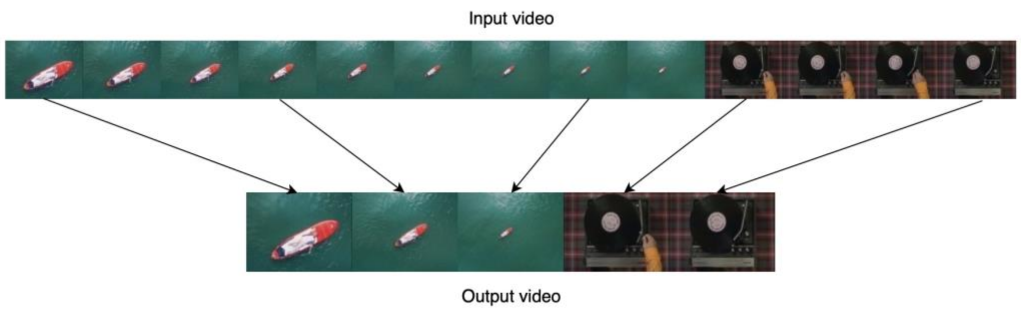


Figure 20. Visualization of a summarized video

Table 3. The video summarizing process evaluation results, with the result lengths being approximately 15% of the original lengths.

Video	Original length	Result length	Ratio of result to original (%)
1	11m27s	1m43s	15
2	14m58s	2m12s	14.7
3	3m44s	35s	15.6
4	3m31s	32s	15.2
5	5m	50s	16.7
6	10m9s	1m39s	16.3
7	10m19s	1m38s	15.8
8	4m57s	45s	15
9	7m14s	1m3s	14.5
10	12m35s	2m3s	16.3
11	6m24s	58s	15.1
12	4m	37s	15.4
13	5m49s	1m	17.2
14	2m24s	27s	18.4
15	2m31s	33s	21.9
16	23m14s	3m36s	15.5

7 CONCLUSION

In conclusion, the main goal of the project was to do research in the field of video summarization and develop a tool that can assist people in improving video summarization workflow by significantly reducing processing time and simplifying video summarizing work.

The project met all of its goals. The developed tool can produce adequate results, with an understandable output that allows people to skim through a video without losing too much information delivered by both video and audio, by using AC-SUM-GAN, which was state-of-the-art research in the field of unsupervised video summarization, as a chosen solution for the processing unit. Furthermore, by utilizing powerful technologies such as Decord, Pydub, and PyTorch, the processing time of video summarizing has been significantly reduced, such that an ordinary user could summarize a video simply by inputting the original video file and receiving a summarized version of it within seconds.

7.1 Future Work

For future improvements, the user interface could be improved. At the moment, the user must interact with the application through the command-line interface, which can be challenging for inexperienced users. For improved usability, a graphical user interface, or a web application could be implemented while providing more scalability. This project could also be packed for the development of other applications in search of new potential use cases.

Furthermore, by focusing on video processing and summarizing, this project opened a new horizon for other AI research topics, such as generating sport highlights, automatically selecting thumbnails, image processing and pattern recognition, and many others.

REFERENCES

1. Statista. Forecast of the number of Youtube users in the World from 2017 to 2025. Accessed 11.02.2022. https://www.statista.com/forecasts/1144088/youtube-users-in-the-world

2. Apostolidis, E., Adamantidou, E., Metsai, A.I., Mezaris, V. and Patras, I., 2021. Video summarization using deep neural networks: A survey. Proceedings of the IEEE, 109(11), pp.1838-1863.

3. IBM Cloud Education. 2020. Artificial Intelligence (AI). Accessed 11.02.2022. https://www.ibm.com/cloud/learn/what-is-artificial-intelligence

4. Ajanki, A., 2018. Differences between machine learning and software engineering. Accessed 11.02.2022. https://futurice.com/blog/differences-between-machine-learning-and-software-engineering

5. Sultan, K., Ali, H. and Zhang, Z., 2018. Big data perspective and challenges in next generation networks. Future Internet, 10(7), p.56.

6. Kavlakoglu E., 2020. AI vs. Machine Learning vs. Deep Learning vs. Neural Networks: What’s the Difference?. Accessed 11.02.2022. https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks

7. Melcher K., 2021. A Friendly Introduction to [Deep] Neural Networks. Accesses 11.02.2022. https://www.knime.com/blog/a-friendly-introduction-to-deep-neural-networks

8. IBM Cloud Education. 2020. Deep Learning. Accessed 11.02.2022. https://www.ibm.com/cloud/learn/deep-learning

9. LeCun, Y. and Bengio, Y., 1995. Convolutional networks for images, speech, and time series. The handbook of brain theory and neural networks, 3361(10), p.1995.

10. IBM Cloud Education. 2020. Convolutional Neural Networks. Accessed 11.02.2022. https://www.ibm.com/cloud/learn/convolutional-neural-networks

11. He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

12. LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P., 1998. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), pp.2278-2324.

13. Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

14. Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems, 25.

15. Ren, S., He, K., Girshick, R. and Sun, J., 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. Advances in neural information processing systems, 28.

16. IBM Cloud Education. 2020. Recurrent Neural Networks. Accessed 11.02.2022. https://www.ibm.com/cloud/learn/recurrent-neural-networks

17. Rassem, A., El-Beltagy, M. and Saleh, M., 2017. Cross-country skiing gears classification using deep learning. arXiv preprint arXiv:1706.08924.

18. Phi M., 2018. Illustrated Guide to LSTM's and GRU's: A step by step explanation. Accessed 11.02.2022. https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

19. Papers With Code. Bidirectional LSTM. Accessed 11.02.2022. https://paperswithcode.com/method/bilstm

20. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. and Bengio, Y., 2014. Generative adversarial nets. Advances in neural information processing systems, 27.

21. TUM Wiki. 2017. Generative Adversarial Networks (GANs). Accessed 23.01.2022. https://wiki.tum.de/pages/viewpage.action?pageId=23562510

22. Johansen A.G., 2020. Deepfakes: What they are and why they're threatening. Accessed 23.01.2022. https://us.norton.com/internetsecurity-emerging-threats-what-are-deepfakes.html

23. Rocca J., 2019. Understanding Variational Autoencoders (VAEs). Accessed 23.01.2022. https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73

24. Kingma, D.P. and Welling, M., 2019. An introduction to variational autoencoders. arXiv preprint arXiv:1906.02691.

25. Xiang Y., 2021. Deploy variational autoencoders for anomaly detection with TensorFlow Serving on Amazon SageMaker. Accessed 23.01.2022. https://aws.amazon.com/blogs/machine-learning/deploying-variational-autoencoders-for-anomaly-detection-with-tensorflow-serving-on-amazon-sagemaker/

26. Larsen, A.B.L., Sønderby, S.K., Larochelle, H. and Winther, O., 2016, June. Autoencoding beyond pixels using a learned similarity metric. In International conference on machine learning (pp. 1558-1566). PMLR.

27. Xian, Y., Sharma, S., Schiele, B. and Akata, Z., 2019. f-vaegan-d2: A feature generating framework for any-shot learning. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 10275-10284).

28. Grondman, I., Busoniu, L., Lopes, G.A. and Babuska, R., 2012. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 42(6), pp.1291-1307.

29. Andrew, A.M., 1999. REINFORCEMENT LEARNING: AN INTRODUCTION by Richard S. Sutton and Andrew G. Barto, Adaptive Computation and Machine Learning series, MIT Press (Bradford Book), Cambridge, Mass., 1998, xviii+ 322 pp, ISBN 0-262-19398-1,(hardback,£ 31.95). Robotica, 17(2), pp.229-235.

30. Pfau, D. and Vinyals, O., 2016. Connecting generative adversarial networks and actor-critic methods. arXiv preprint arXiv:1610.01945.

31. Milind, M.P.M.G.P., 2010. Histogram Based Efficient Video Shot Detection Algorithms. pp.2

32. Wikipedia. Sampling (Signal Processing). Accessed 12.02.2022. https://en.wikipedia.org/wiki/Sampling_(signal_processing)#/media/File:Signal_Sampling.svg

33. GeeksforGeeks, 2022. 0-1 Knapsack Problem | DP-10. Accessed 11.03.2022. https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

34. Python, What is Python? Executive Summary. Accessed 02.04.2022. https://www.python.org/doc/essays/blurb/

35. PyTorch. End-to-end Machine Learning Framework. Accessed 21.02.2022. https://pytorch.org/features/

36. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L. and Desmaison, A., 2019. Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems, 32.

37. NVIDIA. Pytorch. Accessed 21.02.2022. https://www.nvidia.com/en-us/glossary/data-science/pytorch/

38. OpenCV. About. Accessed 21.02.2022. https://opencv.org/about/

39. DMLC. Decord. Accessed 21.02.2022. https://github.com/dmlc/decord

40. Robert J. 2011. Pydub. Accessed 21.02.2022. https://github.com/jiaaro/pydub/

41. Apostolidis, E., Adamantidou, E., Metsai, A.I., Mezaris, V. and Patras, I., 2020. AC-SUM-GAN: Connecting actor-critic and generative adversarial networks for unsupervised video summarization. IEEE Transactions on Circuits and Systems for Video Technology, 31(8), pp.3278-3292.

42. Queudet, A., Abdallah, N. and Chetto, M., 2017. KTS: a real-time mapping algorithm for NoC-based many-cores. The Journal of Supercomputing, 73(8), pp.3635-3651.

43. Song, Y., Vallmitjana, J., Stent, A. and Jaimes, A., 2015. Tvsum: Summarizing web videos using titles. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 5179-5187).