Department of Applied Mathematics and Computer Science
Software Technology

Deep Learning-Based Human Action Recognition

A Multi-Model Approach using CNN and RNN Architectures

Bachelor Thesis

Deep Learning-Based Human Action Recognition
A Multi-Model Approach using CNN and RNN Architectures

Bachelor Thesis
November 25, 2024

By
Bashar Khaled Bdewi

Cover photo: Vibeke Hempler, 2012
Published by: DTU, Software Technology, Richard Petersens Plads, Building 324, DK-2800 Kgs. Lyngby Denmark
www.compute.dtu.dk

Approval

This thesis has been prepared over a period of three months at the Department of Applied Mathematics and Computer Science, Technical University of Denmark (DTU), in partial fulfillment for the degree of Bachelor of Engineering in Software Technology.

It is assumed that the reader has a basic knowledge in the areas of statistics, computer science, and machine learning.

Bashar Khaled Bdewi - s183356

Signature

November 25, 2024

Abstract

This thesis aims to develop a deep learning model for human action recognition in videos, specifically for monitoring regulatory compliance at Novo Nordisk, a leading pharmaceutical company in Denmark. The research addresses the need for automated, accurate, and real-time monitoring of human actions in manufacturing environments to ensure adherence to safety and regulatory protocols.

The proposed model combines Convolutional Neural Networks (CNNs) with Recurrent Neural Networks (RNNs). This hybrid approach leverages the strengths of both architectures: CNNs capture spatial information within video frames utilizing InceptionV3 for frame-level feature extraction, feeding these features into an RNN, particularly Long Short-Term Memory (LSTM) layers, to process and model temporal dependencies and dynamics across frame sequences, enabling robust action recognition in diverse industrial settings.

The model was trained and evaluated on a benchmark action recognition dataset, demonstrating robust performance in classifying human actions critical to regulatory compliance and safety protocols.

The research applies transfer learning techniques for improving model performance, enabling faster and more efficient training. The findings highlight the potential of this approach for automating compliance monitoring in the pharmaceutical industry and broader applications in video surveillance, human-computer interaction, and sports analytics.

Acknowledgements

”Gratitude is the memory of the heart.” – Jean Baptiste Massieu

As I complete this important phase of my academic journey, I wish to express my heartfelt thanks to all the individuals who have supported and guided me throughout this research.

I’m particularly grateful to my supervisor, **Jeppe Revall Frisvad**, from the Department of Applied Mathematics and Computer Science at the Technical University of Denmark (DTU). Your guidance, feedback, and support have been key to shaping this project.

Kristian Dahlmann Jensen, thank you for being such a great advisor during my internship and student work. Your guidance and support really helped me through the tough parts of this important phase.

I am deeply grateful to **Novo Nordisk** for their invaluable support and resources, which have greatly enriched my academic and professional journey.

I would like to express my deepest gratitude to my beloved **family** —my father, my mother, and my siblings—for their unconditional love and encouragement. Your unwavering belief in me has been a constant source of strength and motivation.

To all my **friends** who have been part of my academic journey, I offer my sincere thanks. Your companionship and support have enriched my experience beyond measure.

Lastly, I would like to acknowledge the **Technical University of Denmark** once again for providing the platform and resources that made this thesis possible. The knowledge and skills I have gained here will undoubtedly shape my future endeavors.

Contents

Chapter 1

Introduction

This chapter describes the specific problem addressed by this thesis, the context of the problem, the goal of the project, and outlines the structure of the thesis.

1.1 Background

Videos are becoming mainstream communication channels. According to the 2023 Global Internet Phenomena Report from Sandvine [1]. video data account for more than 65% of all the Internet traffic and are at an annual growth rate of approximately 24%. As a result, such a trend has promoted the development of various video-related applications including online education, remote healthcare, online advertising, virtual reality (VR), etc.

A fundamental technique that contributes to the success of the aforementioned applications is video understanding, which refers to the analysis and interpretation of video contents with advanced algorithms. Traditional video understanding approaches typically aims to extract representative visual features of video clips. However, designing robust and scalable hand-crafted features is challenging, particularly when confronted with complicated appearances and movements.

The prominent success of deep learning in the image domain also inspired researchers to investigate its application in the realm of video understanding. One straightforward idea is to apply the top-performing image models to videos in a frame-by-frame manner. However, neglecting the temporal dimension in modeling poses significant challenges for these methods when analyzing videos that exhibit strong temporal correlations.

For example, while image models can readily recognize the presence of a door in a video sequence, they often struggle to accurately predict whether a person is opening or closing the door.

Additionally, analyzing videos frame by frame will undoubtedly result in a significant increase in computational cost, making it inefficient and impractical for real-world applications [2].

Figure 1.1: Representative Frames in a Video Stream

These challenges have motivated researchers in the field to develop innovative and efficient solutions for various video tasks. The pursuit of such solutions confers immense value and fascination to the study of video understanding.

Video action recognition aims to recognize human actions in a video sequence. As one of the most representative video tasks, video action recognition relies on computer vision and machine learning techniques to first extract representative features from video frames, followed by predicting the action categories, such as running, swimming, or eating.

The video action recognition benchmarks commonly used by researchers have iterated over two generations.

The first generation of datasets are overall smaller in size and shorter in video length, including:

1. UCF101 dataset: This dataset contains a total of 13,320 videos across 101 action categories.
2. HMDB dataset: This dataset consists of 6,766 videos distributed among 51 classes.

By contrast, the second generation of datasets are more challenging due to their larger size and longer duration, which include:

1. Kinetics-400 dataset: This large-scale dataset contains approximately 240,000 video clips covering 400 action classes.
2. Something-something V2 dataset: This fine-grained human activity dataset covers 174 action categories.

1.2 Problem Statement

The need to accurately interpret human actions in videos is particularly vital for applications such as autonomous systems, behavioral analysis, and video retrieval. We need to capture both spatial and temporal features effectively, which are essential for understanding motion and context in videos.

Convolutional Neural Networks (CNNs) have demonstrated success in extracting spatial features, but they do not model the temporal dependencies, which are critical for understanding sequences of actions over time.

On the other hand, Recurrent Neural Networks (RNNs), excel at capturing temporal dynamics but struggle to process complex spatial information in raw video frames.

Thus, the challenge is:

How can we design a model that effectively integrates spatial and temporal information to achieve accurate recognition of predefined human actions in video sequences?

The **primary objective** is to develop and implement a video classification system capable of accurately classifying short videos into predefined categories.

1.3 Methodology

This project leverages a hybrid architecture that utilizes a pretrained Convolutional Neural Network (CNN) model, specifically InceptionV3 model, as a feature extractor to capture spatial information from individual video frames. Subsequently, a Recurrent Neural Network (RNN) with Gated Recurrent Units (GRU) is employed to learn the temporal relationships between the frames. A fully connected layer is then used for the final classification.

This combination of technologies effectively enables the model to handle both the spatial and temporal aspects of video data, which are crucial for achieving accurate video classification.

1.4 Scope and Limitations

The implementation begins with a self-curated dataset designed to emphasize basic methodologies, establishing a foundational understanding and enabling gradual improvements by selecting a few categories from the UCF50 dataset. This choice is driven by limited resources and computational constraints, necessitating a focus on a smaller subset of predefined actions.

We utilize the pretrained InceptionV3 model to emphasize the importance of transfer learning, enhancing computational efficiency and leveraging the performance of this established model.

Advanced sequence models such as Transformers and 3D Convolutional Neural Networks (3D CNNs) are excluded from this study.

Chapter 2

Background

This chapter provides a theoretical background on various concepts [3] essential for understanding the material presented in this thesis.

2.1 Artificial Neural Network

2.1.1 A First Model: The Perceptron

In neural network terminology, a neuron is a parametrized function that takes an input vector (x) and outputs a single value (a). The relationship is defined by the following equation:

$$a = \varphi\left(wx + b\right) \tag{2.1}$$

where the parameters of the neuron are its weights stored in w and a bias term b, and φ is an activation function that is chosen a priori.

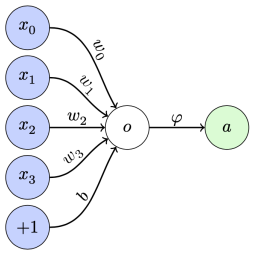

Figure 2.1: The Perceptron

A model made of a single neuron is called a Perceptron. In this model, the predicted output is computed as a linear combination of the input features plus a bias:

$$\hat{y} = \sum_{j=1}^{d} x_j w_j + b \tag{2.2}$$

2.1.2 Multi Layer Perceptrons

To accommodate a broader range of models, neurons can be organized into layers and stacked to form more complex structures. The model described below is referred to as

a multi-hidden-layer model, characterized by additional layers of neurons positioned between the inputs and the output.

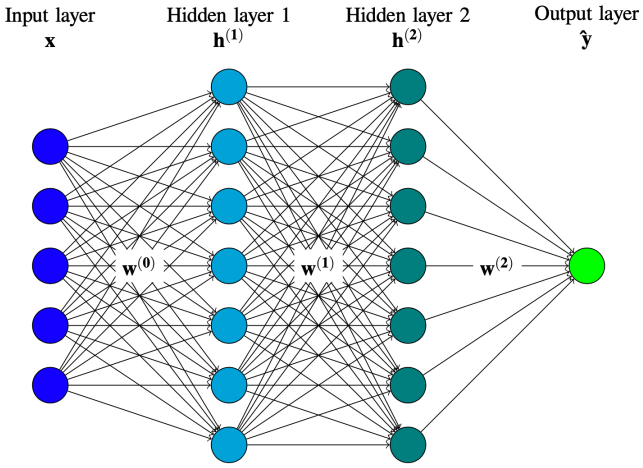

Figure 2.2: Multi Layer Perceptrons (MLP)

The above graphical representation corresponds to the following model:

$$\hat{y} = \varphi_{\text{out}} \left(\sum_i w_i^{(2)} h_i^{(2)} + b^{(2)} \right) \tag{2.1}$$

$$h_i^{(2)} = \varphi \left(\sum_j w_{ij}^{(1)} h_j^{(1)} + b_i^{(1)} \right) \tag{2.2}$$

$$h_i^{(1)} = \varphi \left(\sum_j w_{ij}^{(0)} x_j + b_i^{(0)} \right) \tag{2.3}$$

To be even more precise, the bias terms $b_i^{(l)}$ are not represented in the graphical representation above. Such models with one or more hidden layers are called Multi-Layer Perceptrons (MLP).

In more generality, for the output layer, one might face several situations:

- When regression is at stake, the number of neurons in the output layer is equal to the number of features to be predicted by the model.
- When it comes to classification:
  - In the case of binary classification, the model will have a single output neuron which will indicate the probability of the positive class.
  - In the case of multi-class classification, the model will have as many output neurons as there are classes in the problem.

Once these numbers of input/output neurons are fixed, the number of hidden neurons, as well as the number of neurons per hidden layer, are left as hyper-parameters of the model.

2.1.3 Activation Functions

When designing a Multi-Layer Perceptron (MLP) model for a specific problem, some quantities are predetermined by the problem itself, while others are left as hyperparameters.

Another important hyperparameter in neural networks is the choice of the activation function φ. It is crucial to note that using the identity function as the activation function would result in the MLP merely covering the family of linear models, regardless of its depth. Therefore, in practice, activation functions are chosen that exhibit some linear characteristics but do not behave linearly across the entire range of input values.

Historically, the following activation functions have been proposed:

- **Tanh** (hyperbolic tangent):

$$\tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

- **Sigmoid**:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

- **ReLU** (Rectified Linear Unit):

$$\text{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

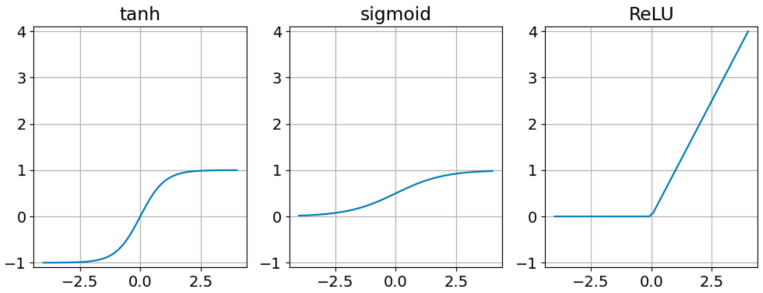

Figure 2.3: Activation Functions

The Special Case of the Output Layer

We might have noticed that in the MLP formulation provided in Equation (1), the output layer has its own activation function, denoted ϕ_{out}. This is because the choice of activation functions for the output layer of a neural network is specifically tailored to the problem at hand.

Indeed, the activation functions discussed in the previous section do not share the same range of output values. It is, therefore, of prime importance to select an adequate activation function for the output layer so that our model outputs values consistent with the quantities it is supposed to predict.

In some scenarios, it would be appropriate to use ReLU (which can output any positive value) as the activation function for the output layer.

As stated earlier, in the case of binary classification, the model will have a single output neuron, and this neuron will output the probability associated with the positive class. This

quantity is expected to lie in the [0, 1] interval, and thus, the sigmoid activation function is the default choice in this setting.

Finally, when dealing with multi-class classification, we have one neuron per output class, and each neuron is expected to output the probability for a given class. In this context, the output values should be between 0 and 1, and they should sum to 1. For this purpose, we use the softmax activation function defined as:

$$\forall i,\ \text{softmax}(o_i) = \frac{e^{o_i}}{\sum_j e^{o_j}}$$

where, for all i, o_i's are the values of the output neurons before applying the activation function.

2.1.4 Loss Functions for MLP Training

We have now introduced the first family of models, the MLP family. To train these models—i.e., to tune their parameters to fit the data—we need to define a loss function to optimize.

Once this loss function is selected, optimization will involve adjusting the model parameters to minimize the loss.

In this section, we will present two standard loss functions: the mean squared error, which is primarily used for regression, and the logistic loss, which is employed in classification settings.

In the following, we assume that we are given a dataset $\mathcal{D}$ made of n annotated samples (x_i, y_i), and we denote the model's output as following:

$$\forall i, \hat{y}_i = m_\theta(x_i)$$

where m_θ is our model and θ is the set of all its parameters (weights and biases).

1. **Mean Squared Error (MSE)**
   The Mean Squared Error is the most commonly used loss function in regression tasks. It is defined as:

   $$\mathcal{L}(\mathcal{D}; m_\theta) = \frac{1}{n}\sum_{i=1}^{n} \|\hat{y}_i - y_i\|^2 = \frac{1}{n}\sum_{i=1}^{n} \|m_\theta(x_i) - y_i\|^2$$

   The quadratic form of MSE heavily penalizes large errors.

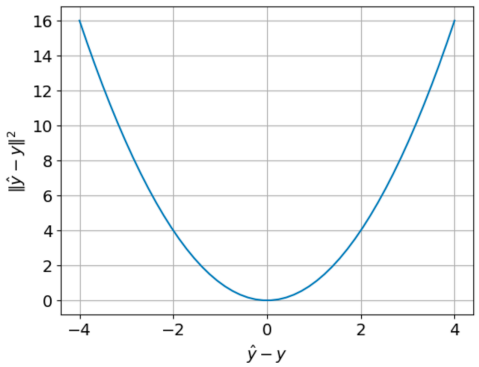

Figure 2.4: Mean Squared Error (MSE)

2. **Logistic Loss**
   The logistic loss is widely used for training neural networks in classification settings. It is defined as:

$$\mathcal{L}(\mathcal{D}; m_\theta) = -\frac{1}{n}\sum_{i=1}^{n} \log p(\hat{y}_i = y_i; m_\theta)$$

   where $p(\hat{y}_i = y_i; m_\theta)$ is the probability predicted by the model m_θ for the correct class y_i. This formulation pushes the model to output a probability of 1 for the correct class as expected.

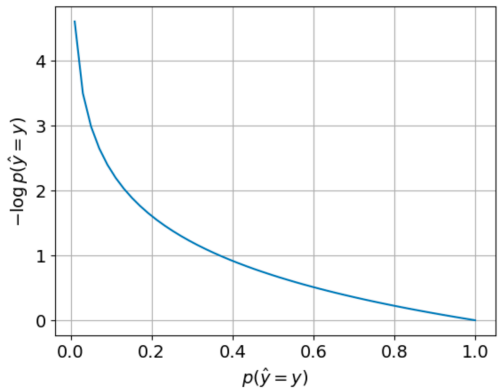

Figure 2.5: Logistic Loss

2.1.5 Optimization

In this section, we will present variants of the Gradient Descent optimization strategy and show how they can be used to optimize neural network parameters.

A. Algorithm 1: Gradient Descent

The Gradient Descent (GD) algorithm is a fundamental method used for finding the minimum of a function. Here, we focus on its application in optimizing the parameters of neural network models. The basic idea behind GD is to update the parameters in the direction that minimally decreases the function, which in the context of machine learning is typically the loss function.

Input: A dataset $\mathcal{D} = (X, y)$

1. Initialize model parameters θ.
2. For $e = 1 \dots E$
   (a) For $(x_i, y_i) \in \mathcal{D}$
      i. Compute prediction $\hat{y}_i = m_\theta(x_i)$.
      ii. Compute gradient $\nabla_\theta \mathcal{L}_i$.
   (b) Compute overall gradient $\nabla_\theta \mathcal{L} = \frac{1}{n}\sum_i \nabla_\theta \mathcal{L}_i$.
   (c) Update parameters θ based on $\nabla_\theta \mathcal{L}$.

The typical update rule for the parameters θ at iteration t is

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \rho \nabla_\theta \mathcal{L},$$

where ρ is an important hyper-parameter of the method, called the learning rate. Essentially, gradient descent updates θ in the direction of the steepest decrease of the loss $\mathcal{L}$.

As one can see in the previous algorithm, when performing gradient descent, model parameters are updated once per epoch, which means a full pass over the whole dataset is required before the update can occur. When dealing with large datasets, this is a strong limitation, which motivates the use of stochastic variants.

B. Algorithm 2: Stochastic Gradient Descent
The idea behind the Stochastic Gradient Descent algorithm is to obtain inexpensive estimates for the gradient of the loss function over the entire dataset, denoted as:

$$\nabla_\theta \mathcal{L}(\mathcal{D}; m_\theta) = \frac{1}{n} \sum_{(x_i, y_i) \in \mathcal{D}} \nabla_\theta \mathcal{L}(x_i, y_i; m_\theta),$$

where $\mathcal{D}$ represents the whole training set. To achieve this, subsets of data, called minibatches, are used, and the gradient computed on these minibatches:

$$\nabla_\theta \mathcal{L}(\mathcal{B}; m_\theta) = \frac{1}{b} \sum_{(x_i, y_i) \in \mathcal{B}} \nabla_\theta \mathcal{L}(x_i, y_i; m_\theta)$$

serves as an estimator for $\nabla_\theta \mathcal{L}(\mathcal{D}; m_\theta)$. This results in the following algorithm, where parameter updates occur after processing each minibatch, multiple times per epoch.

Input: A dataset $\mathcal{D} = (X, y)$

1. Initialize model parameters θ.
2. For $e = 1 \dots E$
   (a) For $t = 1 \dots n_{\text{minibatches}}$
      i. Draw minibatch $\mathcal{B}$ as a random sample of size b from $\mathcal{D}$.
      ii. For $(x_i, y_i) \in \mathcal{B}$
         A. Compute prediction $\hat{y}_i = m_\theta(x_i)$.
         B. Compute gradient $\nabla_\theta \mathcal{L}_i$.
      iii. Compute minibatch-level gradient $\nabla_\theta \mathcal{L}_\mathcal{B} = \frac{1}{b} \sum_i \nabla_\theta \mathcal{L}_i$.
      iv. Update parameters θ based on $\nabla_\theta \mathcal{L}_\mathcal{B}$.

As a consequence, when using SGD, parameter updates are more frequent but ”noisy” since they are based on a minibatch estimation of the gradient rather than relying on the true gradient.

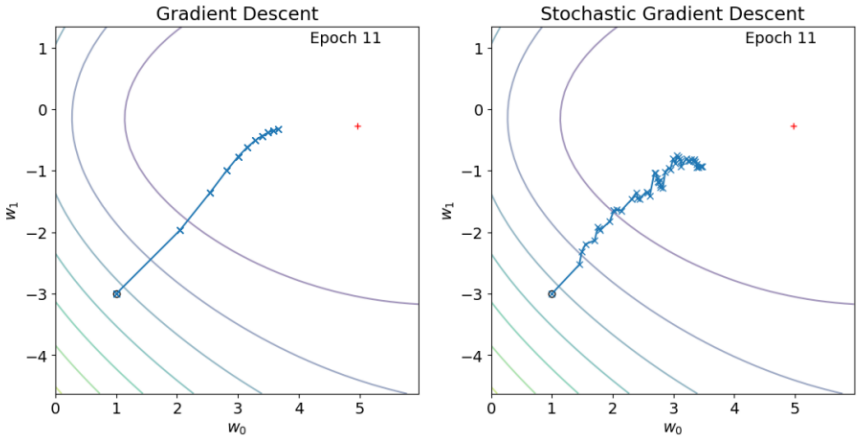

Figure 2.6: Gradient Descent VS. Stochastic Gradient Descent

Apart from facilitating more frequent parameter updates, SGD offers an additional benefit crucial for optimizing neural networks. Contrary to the scenario in the Perceptron case, as illustrated below, the Mean Squared Error (MSE) loss—and similarly, the logistic loss—is no longer convex in the model parameters when the model includes at least one hidden layer.

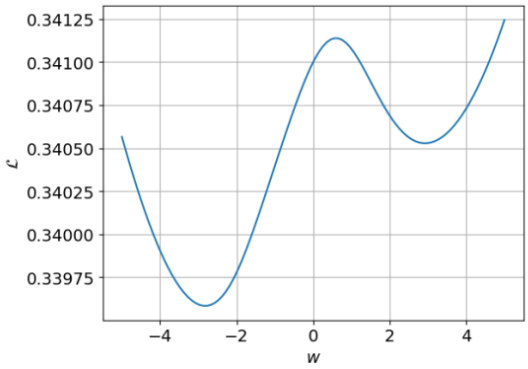

Figure 2.7: Non-Convexity

Gradient Descent is known to suffer from local optima, which pose a serious problem in loss landscapes for this optimization method. On the other hand, Stochastic Gradient Descent can benefit from its noisy gradient estimations, which help it escape local minima.

C. Algorithm 3: Adam Optimization Algorithm
Adam is a variant of the Stochastic Gradient Descent method. It incorporates modifications in the steps performed during each parameter update. Firstly, Adam utilizes what is known as momentum, which essentially involves using past gradient updates to smooth the trajectory in parameter space during optimization. An interactive illustration of momentum can be found in Goh, 2017. The modified gradient, using momentum, is defined as:

$$m(t+1) \leftarrow \frac{1}{1-\beta_1^t}\left[\beta_1 m(t) + (1-\beta_1)\nabla_\theta \mathcal{L}\right]$$

where β_1 is the momentum decay factor. When β_1 is zero, we have $m(t+1) = \nabla_\theta \mathcal{L}$, and for $\beta_1 \in (0,1)$, $m(t+1)$ balances the current gradient estimate with information about past estimates stored in $m(t)$.

Another significant aspect of Adam is its use of an adaptive learning rate. Unlike traditional SGD, which uses a constant learning rate ρ for all model parameters, Adam adjusts the learning rate for each parameter θ_i individually, defined as:

$$\hat{\rho}_{(t+1)}(\theta_i) = \frac{\rho}{\sqrt{s_{(t+1)}(\theta_i)} + \epsilon}$$

where ϵ is a small constant ensuring numerical stability, and $s_{(t+1)}(\theta_i)$, which also uses momentum, is given by:

$$s_{(t+1)}(\theta_i) = \frac{1}{1-\beta_2^t}\left[\beta_2 s_{(t)}(\theta_i) + (1-\beta_2)(\nabla_{\theta_i}\mathcal{L})^2\right]$$

This mechanism lowers the learning rate for parameters that have experienced large updates in past iterations, hence adapting to the specific needs of each parameter.

The overall update rule for Adam is:

$$\theta_{(t+1)} \leftarrow \theta_{(t)} - \hat{\rho}_{(t+1)}(\theta) m_{(t+1)}$$

D. The Curse of Depth: Challenges in Deep Neural Networks
One should note that weights further from the model's output inherit gradient rules composed of more terms. Consequently, when some of these terms become increasingly smaller, there is a higher risk for those weights that their gradients will collapse to zero. This is known as the vanishing gradient effect, a common phenomenon in deep neural networks.

2.1.6 Regularization
One of the strengths of neural networks is their ability to approximate any continuous function given a sufficient number of parameters. This capability classifies them as universal approximators in machine learning settings. However, an important risk associated with universal approximators is overfitting the training data. More formally, given a training dataset $\mathcal{D}_t$ drawn from an unknown distribution $\mathcal{D}$, model parameters are optimized to minimize the empirical risk:

$$\mathcal{R}_e(\theta) = \frac{1}{|\mathcal{D}_t|}\sum_{(x_i,y_i)\in\mathcal{D}_t} \mathcal{L}(x_i, y_i; m_\theta)$$

whereas the real objective is to minimize the "true" risk:

$$\mathcal{R}(\theta) = E_{x,y\sim\mathcal{D}}[\mathcal{L}(x, y; m_\theta)]$$

and it is noted that both objectives do not have the same minimizer.

To avoid this pitfall, it is advisable to employ regularization techniques, as will be presented in the following sections.

1. **Early Stopping**
   It can be observed that training a neural network for a too large number of epochs can lead to overfitting. The true risk is estimated through the use of a validation set that is not seen during training.

   The whole idea behind the "early stopping" strategy consists in stopping the learning process as soon as the validation loss stops improving. However, when the validation loss tends to oscillate, one often waits for several epochs before assuming that the loss is unlikely to improve in the future. The number of epochs to wait is called the patience parameter.

2. **Loss Penalization**
   Another important method to enforce regularization in neural networks is through loss penalization. A typical example of this regularization strategy is L2 regularization. If we denote by $\mathcal{L}_r$ the L2-regularized loss, it can be expressed as:

$$\mathcal{L}_r(\mathcal{D}; m_\theta) = \mathcal{L}(\mathcal{D}; m_\theta) + \lambda \sum_\ell \|\theta^{(\ell)}\|_2^2$$

   where $\theta^{(\ell)}$ is the weight matrix of layer ℓ. This type of regularization tends to reduce the magnitude of large parameter values during the learning process, which is known to help improve generalization.

3. **DropOut**
   The idea behind Dropout is to switch off some of the neurons during training. The switched off neurons change at each mini-batch such that, overall, all neurons are trained during the whole process.

   The concept is very similar in spirit to a strategy that is used for training random forests, which consists in randomly selecting candidate variables for each tree split inside a forest, which is known to lead to better generalization performance for random forests. The main difference here is that one can not only switch off input neurons but also hidden-layer ones during training.

2.2 Convolutional neural network

Convolutional Neural Networks (CNNs) was primarily used for handwritten character recognition. CNNs require fewer parameters than fully connected networks, making them superior to their peer deep networks with easier parallel computation on GPUs. Over time, CNNs have dominated a wide range of tasks such as image classification, object detection, semantic segmentation, and instance segmentation.

In recent years, researchers extend CNNs' ability to applications including natural language processing, recommendation systems, etc [2].

This section is organized as below: First, we will explain convolutions in neural networks, followed by an introduction of pooling, which is a common operation in CNNs; lastly, we will present the detailed architecture of InceptionV3 model.

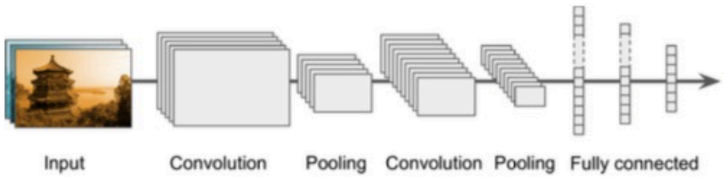

Figure 2.8: Convolutional Neural Network (CNN)

2.2.1 Convolution

Convolution is a fundamental mathematical operation used extensively in signal processing and image processing. In image processing, a two-dimensional convolution is widely used since images are two-dimensional structures. It involves convolving a filter kernel with an input image to produce a feature map. The output is computed by summing the element-wise product of the filter kernel and a region of the input image.

Given an image and a filter, the convolution operation in the input image **X** and the filter **W** is defined as follows:

$$\mathbf{Y} = \mathbf{W} * \mathbf{X} \tag{2.2}$$

where **X** is the input and **W** is the kernel function. The output **Y** is also referred to as the feature map, and the convolution operation is denoted by $*$.

To improve the flexibility of feature extraction and increase the diversity of convolution, the standard convolution operation can be supplemented with a sliding stride for the convolution kernel and a zero-padding operation. This additional step allows for a wider range of features to be extracted and enhances the effectiveness of the convolution operation.

Finally, the size of the feature map can be calculated using a formula that takes into account the input size, filter size, stride, and padding:

$$o = \left\lfloor \frac{n + 2p - f}{s} \right\rfloor + 1 \tag{2.3}$$

where o is the height/width of the feature map, f is the height/width of the filter conducting the convolution operation, n denotes the input's height/width, and the stride and padding sizes are denoted as s and p, respectively [2].

2.2.2 Pooling

The pooling function explores the statistical properties of neighboring regions of a specific location and uses the derived statistics to represent that all elements in that region. For instance, the max pooling function selects the highest value in a neighboring rectangular region.

Other frequently used pooling functions involve the average value of adjacent rectangular areas, and the weighted average function based on the distance from the center pixel. Convolution operations are effective in decreasing the number of connections in a network, but they do not significantly reduce the number of units in the feature map.

Directly attaching a classifier to the feature map can result in dense weight matrix, making the model prone to overfitting. To solve this problem, a pooling function can be added

after the convolution operations, so as to further reduce feature dimension and thus avoid overfitting [2].

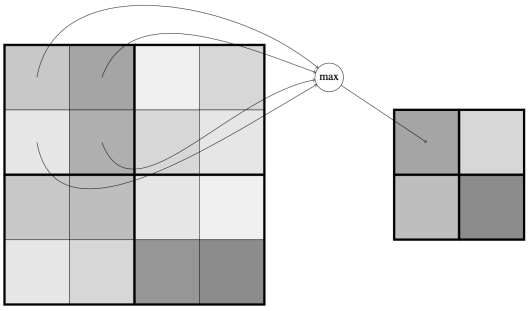

Figure 2.9: The pooling Operation

2.2.3 Classic Convolutional Neural Networks

Several CNN models form the foundation of contemporary deep learning architectures and continue to be used as benchmarks for evaluating performance.

We will begin by discussing GoogLeNet as a starting point, followed by InceptionV3, which builds upon it with improved architectural refinements and optimizations.

Both GoogLeNet and InceptionV3 use Inception blocks, which consist of parallel convolutional filters of different sizes to capture multi-scale features. While GoogLeNet introduced the original Inception block, InceptionV3 refined it with additional techniques like factorized convolutions and grid reduction to improve efficiency and accuracy, making it a more advanced and computationally optimized model [2].

1. GoogLeNet (InceptionV1)

GoogLeNet gained prominence in the 2014 ImageNet image classification competition. It addresses the challenge of determining the optimal convolution kernel size. Previous CNNs used kernels that varied from 1×1 to 11×11, with no consensus on the best size. GoogLeNet introduced a novel architecture, the Inception block, as illustrated in Fig. 2.10.

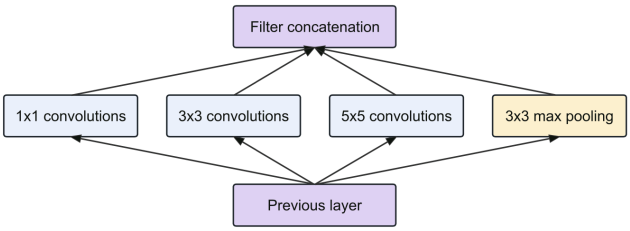

Figure 2.10: Inception Block

With four parallel paths, the Inception block enlarges the capacity of CNNs while maintaining an affordable computational cost. In this way, GoogLeNet takes advantage of several different-sized kernels in one model.

Specifically, the Inception block has four parallel paths:

- The first three paths apply convolutions with filter sizes of 1×1, 3×3, and 5×5 to capture information at different spatial scales.

- The two middle paths use 1×1 convolutions to reduce the number of channels and thus the model's computational complexity.

- The fourth path uses max pooling followed by a 1×1 convolution to change the number of channels.

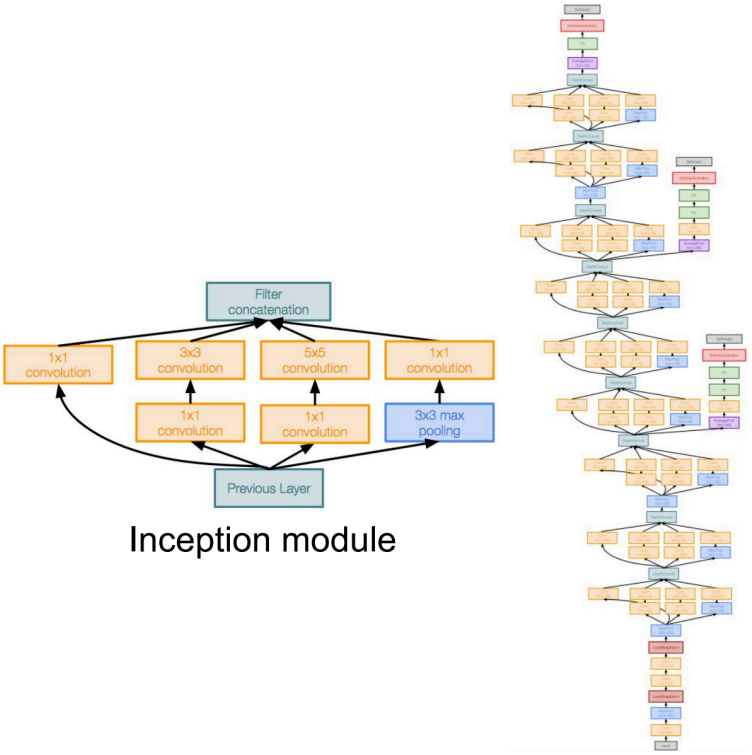

Figure 2.11: GoogLeNet Model Architecture [4] - Main Inception Module

All four paths use appropriate padding to match input and output sizes, and their outputs are concatenated along the channel dimension to form the block output. In practice, the number of output channels per layer in the Inception block is a key hyperparameter that is often tuned to achieve the best performance [2].

2. InceptionV3

InceptionV3, a deep CNN architecture, was pre-trained on the ImageNet dataset and has a total of 42 layers, and its architectural improvements result in a lower error rate compared to earlier versions like GoogLeNet (InceptionV1). This architecture is particularly useful for multi-scale feature extraction, enabling it to capture complex patterns across different frames.

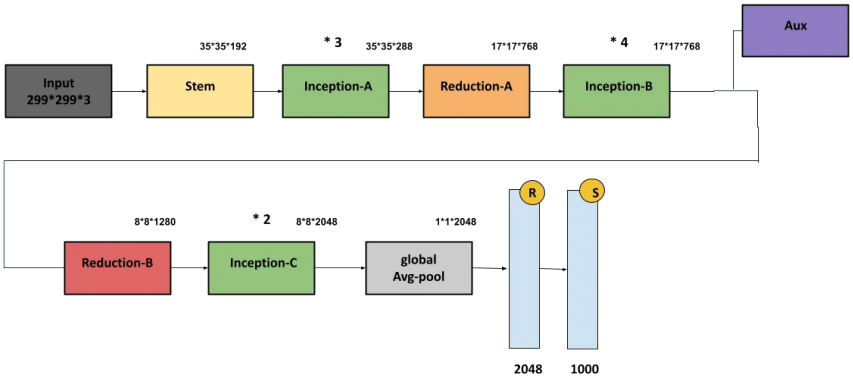

Figure 2.12: InceptionV3 Model Architecture

Key modifications in InceptionV3 include:

- **Factorization into Smaller Convolutions**: Breaking down large convolutional filters into smaller ones to reduce computational cost and improve efficiency.
- **Spatial Factorization into Asymmetric Convolutions**: Replacing square convolutions (e.g., 3×3) with asymmetric convolutions (e.g., 1×3 followed by 3×1) to further reduce computation.
- **Utility of Auxiliary Classifiers**: Adding auxiliary classifiers in intermediate layers to enhance gradient flow and prevent vanishing gradients, aiding in faster and more stable training.
- **Efficient Grid Size Reduction**: Using specialized techniques to reduce the grid size without losing important spatial information, allowing for deeper networks without excessive computation.

Here's a brief explanation of each component in the InceptionV3 architecture [5]:

1. Stem Block IT serves as the initial stage of the network, where raw input images are processed and performs initial convolution and pooling operations to reduce spatial dimensions and prepare data for further stages.

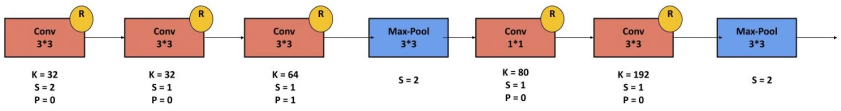

Figure 2.13: Stem Block

2. Inception-A Block It captures multi-scale features with a range of convolutional filters.It utilizes parallel 1×1, 3×3, and 5×5 convolutions to learn features at different spatial resolutions.

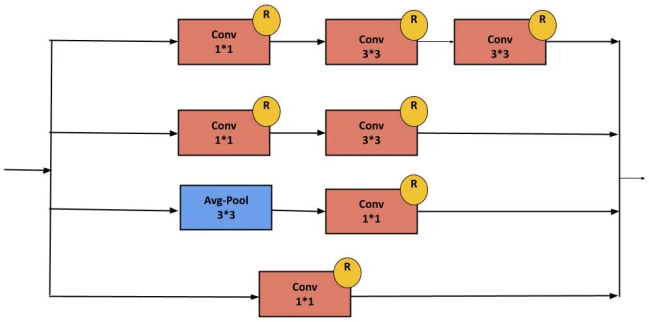

Figure 2.14: Inception-A Block

3. Inception-B Block It reduces computation by splitting convolutions into smaller, asymmetric operations. It applies 1×7 and 7×1 convolutions to capture features efficiently, enhancing depth and complexity with reduced computational load.

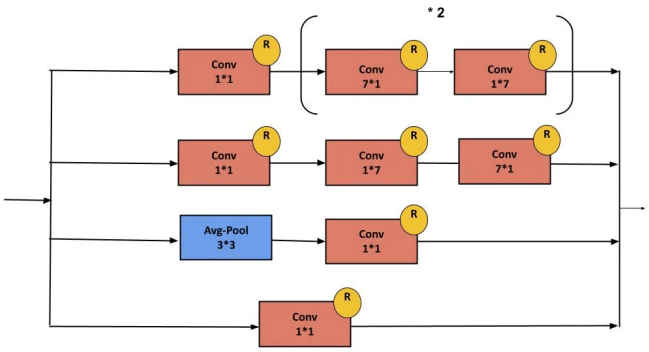

Figure 2.15: Inception-B Block

4. Inception-C Block It captures more complex patterns by combining smaller convolutional filters and uses 1×1, 3×3, and 1×3 convolutions in parallel paths to allow the model to learn diverse features.

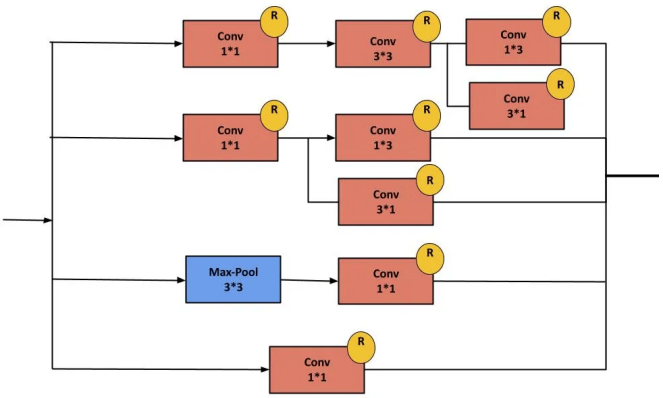

Figure 2.16: Inception-C Block

5. Reduction-A Block It reduces spatial dimensions, which helps manage memory and computation and combines pooling and convolution operations to down-sample while retaining essential information.

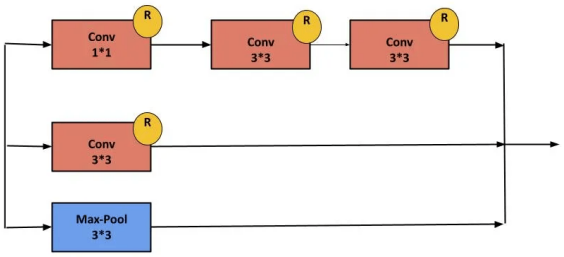

Figure 2.17: Reduction-A Block

6. Reduction-B Block It further reduces spatial dimensions later in the network. It uses a combination of 3×3 and 1×1 convolutions with pooling to down-sample and reduce complexity before the fully connected layers.

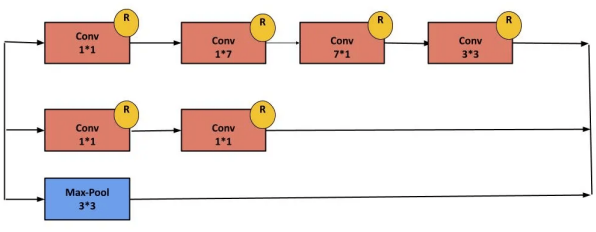

Figure 2.18: Reduction-B Block

7. Auxiliary Classifier Block It acts as a side branch to help the model converge faster and address vanishing gradients. It provides an intermediate output during training, adding an auxiliary loss to guide learning, especially in deeper networks.

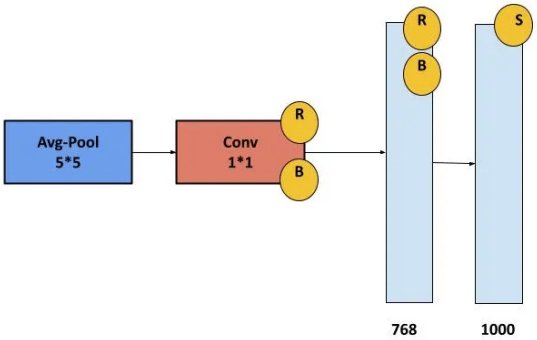

Figure 2.19: Auxiliary Classifier Block

In this project, we load the InceptionV3 model with pretrained weights and remove the final classification layer.

2.3 Recurrent Neural Network

The CNN architecture discussed above exploits convolution for efficient spatial (2D) information extraction, such as images. Considering domains that involve sequential data, CNNs might not be the optimal choice. In tasks such as video understanding, RNNs are the preferred models. In particular, RNNs are designed to capture the dynamic correlations of sequential data via recurrent connections, which use state variables to memorize previous information and determine current output in combination with current input [2].

Recurrent neural networks (RNNs) proceed by processing elements of a time series one at a time. Typically, at time t, a recurrent block will take both the current input x_t and a hidden state h_{t-1} that aims at summarizing the key information from past inputs $\{x_0, \ldots, x_{t-1}\}$, and will output an updated hidden state h_t:

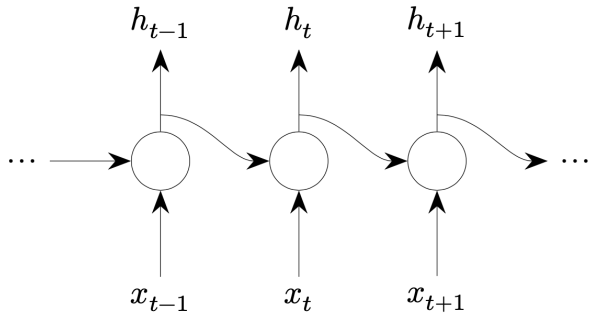

Figure 2.20: Recurrent Neural Network (RNN)

RNNs perform differently from a fully connected network. At the time step t, a typical block in RNNs will take as input an external input vector $\mathbf{x}(t) \in R^n$ and a hidden state $\mathbf{h}(t-1) \in R^r$ generated at the previous time step $t-1$ to calculate the current hidden state $\mathbf{h}(t) \in R^r$. Then the current hidden state will be used to generate the output vector $\mathbf{z}(t) \in R^m$:

$$\mathbf{h}(t) = \sigma\left(\mathbf{W}_x\mathbf{x}(t) + \mathbf{W}_h\mathbf{h}(t-1) + \mathbf{b}_h\right)$$
$$\mathbf{z}(t) = \text{softmax}\left(\mathbf{W}_z\mathbf{h}(t) + \mathbf{b}_z\right),$$

where $\mathbf{W}_x \in R^{r\times n}$, $\mathbf{W}_h \in R^{r\times r}$, and $\mathbf{W}_z \in R^{m\times r}$ are weight matrices, while $\mathbf{b}_h$ and $\mathbf{b}_z$ denote biases. The function σ defines the sigmoid function:

$$\sigma(x) = \frac{1}{1+e^{-x}},$$

whereas softmax represents the softmax function.

Traditional RNNs endure two major problems: One is their limitation in modeling long-term dependencies and the other is storage limitation, hindering RNNs from approximating sequences of arbitrary complexity. In addition, vanishing and exploding gradients further impede RNNs' real-world applications.

The vanishing gradient means that as gradients propagate back through time, they might be shrunk exponentially during the process. The exploding gradient problem occurs when gradients become too large during training, making it difficult to train the network effectively. To tackle the barriers above, Long Short-Term Memory models are proposed as one of the solutions [2].

Long Short-Term Memory (LSTM)

LSTM is a type of Recurrent Neural Network (RNN) that addresses the challenge of storing and accessing information over long time sequences. LSTM introduces non-linear multiplicative gates and a memory cell. The non-linear gates, including the input, output, and forget gates, control the flow of information into and out of the memory cell [2].

LSTM unit takes an external vector $\mathbf{x}(t) \in R^n$ as input and maps it to an output vector $\mathbf{z}(t) \in R^m$ by computing the activations of its units with the following equations recursively from $t = 1$ to $t = T$:

$$i(t) = \sigma\left(\mathbf{W}_{xi}\mathbf{x}(t) + \mathbf{W}_{hi}\mathbf{h}(t-1) + \mathbf{W}_{ci}\mathbf{c}(t) + \mathbf{b}_i\right), \tag{2.3}$$
$$f(t) = \sigma\left(\mathbf{W}_{xf}\mathbf{x}(t) + \mathbf{W}_{hf}\mathbf{h}(t-1) + \mathbf{W}_{cf}\mathbf{c}(t) + \mathbf{b}_f\right), \tag{2.4}$$
$$c(t) = f(t)\cdot\mathbf{c}(t-1) + i(t)\cdot\tanh\left(\mathbf{W}_{xc}\mathbf{x}(t) + \mathbf{W}_{hc}\mathbf{h}(t-1) + \mathbf{b}_c\right), \tag{2.5}$$
$$o(t) = \sigma\left(\mathbf{W}_{xo}\mathbf{x}(t) + \mathbf{W}_{ho}\mathbf{h}(t-1) + \mathbf{W}_{co}\mathbf{c}(t) + \mathbf{b}_o\right), \tag{2.6}$$
$$h(t) = o(t)\cdot\tanh\left(\mathbf{c}(t)\right), \tag{2.7}$$

where $\mathbf{x}(t)$ and $\mathbf{h}(t)$ represent the input and hidden vectors at time step t, respectively. The activation vectors for the input gate, forget gate, memory cell, and output gate are denoted by $i(t)$, $f(t)$, $c(t)$, and $o(t)$, respectively.

The weight matrix between two vectors α and β is represented by $\mathbf{W}_{\alpha\beta}$. For instance, $\mathbf{W}_{xi}$ corresponds to the weight matrix from input $\mathbf{x}(t)$ to the input gate $i(t)$.

At each time step t, an LSTM unit takes the input $\mathbf{x}(t)$ and the previous hidden state $\mathbf{h}(t-1)$ as input. The information stored in the memory cell is determined by the information in the previous memory cell unit $\mathbf{c}(t-1)$ and the activation of the input gate $i(t)$. Specifically, LSTM has three main steps:

1. To forget previous state's information according to the forget gate. In this step, $f(t)$ serves as the forget gate.
2. To update the information of the current state. The current state of the memory cell is obtained by summing up previous information and the current input signal.
3. To output information. The output gate controls the information received by the hidden state variable $\mathbf{h}(t)$.

To summarize, LSTM is capable of long-term temporal memory with carefully designed memory units and gates. As LSTM discards some long-term information and is required to memorize only partial information, it avoids the issues of vanishing and exploding gradients.

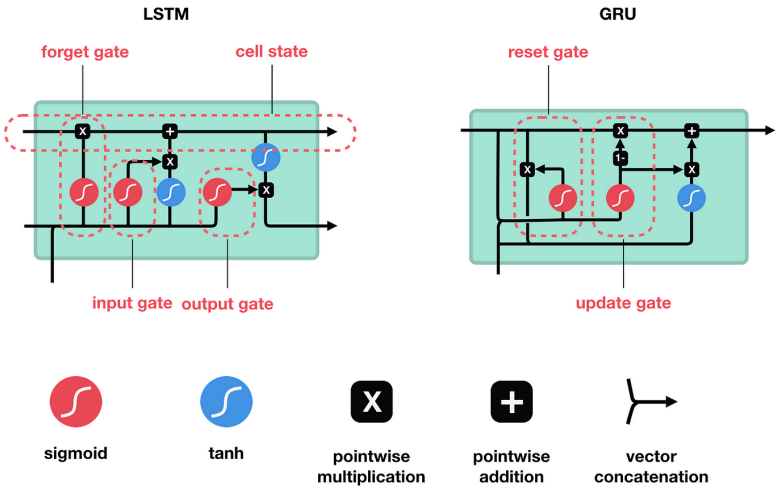

Figure 2.21: GRU vs LSTM [6]

Gated Recurrent Units (GRUs)

GRUs [7] are a variation of the LSTM. It combines the forget and input gates into a single update gate. It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models, and has been growing increasingly.

Like LSTM, GRU is designed to model sequential data by allowing information to be selectively remembered or forgotten over time. However, GRU has a simpler architecture than LSTM, with fewer parameters, which can make it easier to train and more computationally efficient.

The main difference between GRU and LSTM is the way they handle the memory cell state. In LSTM, the memory cell state is maintained separately from the hidden state and is updated using three gates: the input gate, output gate, and forget gate. In GRU, the memory cell state is replaced with a *candidate activation vector*, which is updated using two gates: the **reset gate** and **update gate**.

- The **reset gate** determines how much of the previous hidden state to forget.
- The **update gate** determines how much of the candidate activation vector to incorporate into the new hidden state.

Overall, GRU is a popular alternative to LSTM for modeling sequential data, especially in cases where computational resources are limited or where a simpler architecture is desired.

Like other recurrent neural network architectures, GRU processes sequential data one element at a time, updating its hidden state based on the current input and the previous hidden state. At each time step, the GRU computes a *candidate activation vector* that combines information from the input and the previous hidden state. This candidate vector is then used to update the hidden state for the next time step.

The candidate activation vector is computed using two gates: the **reset gate** and the **update gate**.

- The **reset gate** determines how much of the previous hidden state to forget.
- The **update gate** determines how much of the candidate activation vector to incorporate into the new hidden state.

The GRU architecture is defined by the following equations:

1. The reset gate r_t and update gate z_t are computed using the current input x_t and the previous hidden state h_{t-1}:

$$r_t = \sigma(W_r[h_{t-1}, x_t])$$

$$z_t = \sigma(W_z[h_{t-1}, x_t])$$

   where W_r and W_z are weight matrices that are learned during training, and σ represents the sigmoid activation function.

2. The candidate activation vector $\tilde{h}_t$ is computed using the current input x_t and a modified version of the previous hidden state h_{t-1}, scaled by the reset gate r_t:

$$\tilde{h}_t = \tanh(W_h[r_t * h_{t-1}, x_t])$$

   where W_h is another weight matrix, and tanh represents the hyperbolic tangent activation function.

3. The new hidden state h_t is computed by combining the candidate activation vector $\tilde{h}_t$ with the previous hidden state h_{t-1}, weighted by the update gate z_t:

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

The reset gate determines how much of the previous hidden state to remember or forget, while the update gate determines how much of the candidate activation vector to incorporate into the new hidden state. The result is a compact architecture that is able to selectively update its hidden state based on the input and previous hidden state, without the need for a separate memory cell state like in LSTM.

Chapter 3

Methodology

The purpose of this chapter is to provide an overview of the Methodological research and Workflow that is used in this project.

3.1 Research method

The convolutional-based methods mainly focus on classifying short video clips and are limited in handling excessively long untrimmed videos, which poses a challenge for real-world applications that require classification of events in long videos. Many of these works have to sample many frames in different time steps and apply average pooling or maximum pooling on pre-extracted segmental features to solve the challenge of long videos.

However, it could often lead to sub-optimal performance in downstream tasks due to a lack of proper long-term temporal modeling. To address this issue, the exploration of advanced long-range temporal feature aggregation techniques is needed. Feature aggregation is the crucial step in the analysis of video data, where multiple features are combined into a compact representation to reduce feature redundancy.

In the context of video action recognition, proper aggregation of the long-range temporal features plays a key role in capturing the dynamics of actions. As such, there has been a growing interest in the development of long-range temporal feature aggregation methods to capture the temporal continuity of actions or events in videos [2].

Recurrent Neural Networks (RNNs) have become one of the mainstream approaches in the field of video recognition since the RNN-based methods contain powerful abilities to process sequential data and handle long-term temporal dependencies.

Long Short-Term Memory (LSTM) and Gate Recurrent Unit (GRU) are widely used RNN models to model the temporal dynamics in videos due to their ability to avoid the severe “vanishing gradient” effect.

Designing LSTM networks demonstrated that the joint finetuning of the CNN models and LSTMs resulted in a slight improvement in performance compared to using only the LSTM model.

As a result, by combining the outputs of CNN and LSTM models to jointly model spatial–temporal cues, finding them to be highly complementary.

In addition, it was verified that using LSTM or GRU as temporal aggregation modules can achieve good video classification performance on the large-scale video understanding dataset [2].

In this project, we employ *Temporal Aggregation with Recurrent Neural Networks* to capture sequential dependencies across frames, enhancing the model's ability to understand and accurately classify dynamic actions.

3.2 Model Architecture

The architecture applied in the project, as mentioned earlier, leverages a pretrained CNN (InceptionV3) as a feature extractor for the video frames and an RNN (LSTM or GRU) to learn the temporal relationships between frames. This combination allows the model to effectively handle both the spatial and temporal aspects of the video data.

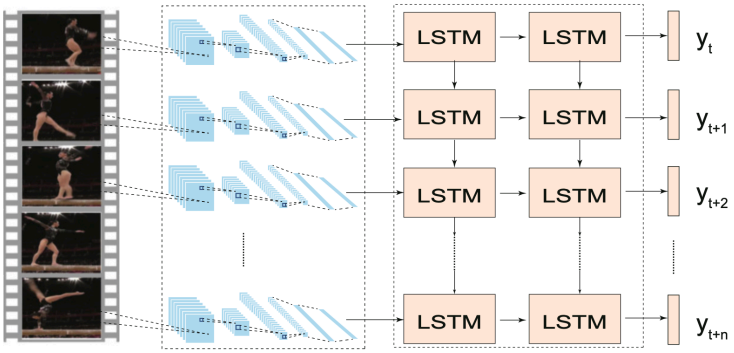

Figure 3.1: CNN - RNN Model Architecture

3.2.1 Input Data

Videos are loaded into memory frame by frame, each frame is initially cropped to a square around the center and then resized to match the input size expected by InceptionV3, which is 224×224 pixels. As a result the shape of each frame, after cropping and resizing, becomes $(224, 224, 3)$, and the shape of the entire video tensor would be $(20, 224, 224, 3)$—20 frames, each with dimensions $(224, 224, 3)$.

3.2.2 Features Extraction

The preprocessed frames are then fed into InceptionV3, which consists of a series of convolutional layers that progressively alter the input tensor's dimensions. These convolutions, combined with max-pooling layers, initially reduce the spatial dimensions while increasing the depth (number of channels or features) to control computational complexity.

Typically, after the first few layers, an input frame of $(224, 224, 3)$ is reduced to smaller feature maps of shape $(112, 112, 64)$. As it progresses through the network, the spatial dimensions continue to decrease while the depth (number of channels) increases, enhancing the model's capacity to capture complex spatial patterns.

The feature map is then reduced further to become of shape $(5, 5, 2048)$.

The GlobalAveragePooling layer reduces this feature map to $(1, 1, 2048)$ by computing the average value of each feature map channel resulting in a 2048-1D dimensional feature vector that serves as a spatial compact representation of the original frame.

For an entire video sequence, assuming `MAX SEQUENCE LENGTH` is 20, the output tensor shape is $(20, 2048)$, representing 20 frames, each encoded as a 2048-dimensional feature vector.

In Summary:

- Input Shape for Each Frame: $(224, 224, 3)$
- Intermediate Feature Map Shapes: from $(112, 112, 64)$ to $(5, 5, 2048)$.
- Final Feature Vector for Each Frame: $(2048,)$
- Final Feature Tensor for each Video Sequence: $(20, 2048)$.

This tensor of feature vectors (one per frame) will then be processed by a recurrent neural network (RNN) to capture temporal features before making the final classification.

3.2.3 Sequence Modeling

GRU layers use a gating mechanism (reset and update gates) to manage information flow, enabling them to capture dependencies over varying time scales. The output from the GRU at each time step is influenced by the current input and the previous hidden state, producing an updated hidden state.

Assuming that the BATCH_SIZE is 64, the input shape is $(64, 20, 2048)$. The GRU processes each 2048-dimensional feature vector sequentially over the 20 time steps.

If the GRU layer has 16 units, it transforms each input feature vector into a 16-dimensional hidden state vector for each input sequence. Thus, each of the 20 time steps produces a 16-dimensional output vector.

The final GRU layer outputs only the **final hidden state** for the entire sequence (a single output per sequence). This results in an output of shape $(64, 8)$, where each 8-dimensional vector represents the aggregated temporal information for one video.

In Summary

- Input Shape: $(64, 20, 2048)$.
- Intermediate Output Shape: $(64, 20, 16)$, where each frame in the sequence is represented by a 16-dimensional output vector.
- Final Output Shape: $(64, 8)$, where each video sequence is represented by a single 8-dimensional feature vector.

In essence, the GRU compresses the sequential information into a single feature vector, effectively capturing the temporal dependencies and patterns within the input video.

3.2.4 Final Classification

The output tensor from the GRU is $(64, 8)$, which is passed to the Dense layer.

The Dense layer is designed to map the 8-dimensional input to the number of classes for classification. If we have 3 classes, the Dense layer's output will have the shape

$$(64, 3).$$

This final output shape represents the predicted class probabilities for each input video sequence. The predictions can be obtained using a softmax activation function applied in the Dense layer, which converts the output logits into probabilities for each class.

The following table summarizes the tensor shape transformation through the whole network:

Layer	Output Shape
Input	(Max Sequence Length, Height, Width, RGB)
Initial InceptionV3	(Max Sequence Length, Height, Width, Number of Features)
Intermediate InceptionV3	(Max Sequence Length, Height, Width, Number of Features)
Final InceptionV3	(Batch Size, Max Sequence Length, Number of Features)
GRU (`Initial hidden states`)	(Batch Size, Max Sequence Length, Number of Hidden States)
GRU (`Final hidden states`)	(Batch Size, Number of Hidden States)
Dense	(Batch Size, Number of Classes)

Table 3.1: Tensor shape progression through the network layers

By following these steps, we can trace how the input video data is transformed through the layers of the network, including the InceptionV3 and GRU phases, leading to the final classification output.

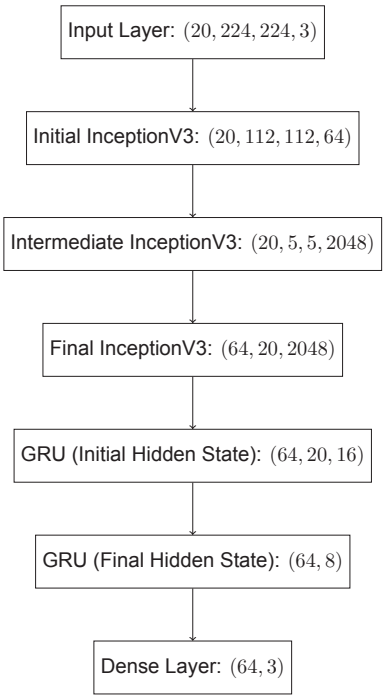

Figure 3.2: Transformation of Tensor Shapes Through the Network Layers

Chapter 4

Video classification implementation

This chapter presents a comprehensive overview of the dataset preparation process, outlines the steps taken to ensure the dataset is suitable for training and evaluating the proposed model, highlighting the significance of data quality and representation in the context of human action recognition. Additionally, it outlines the procedures and techniques employed to develop an effective system for human action recognition.

4.1 Overview of the Dataset

The UCF50 dataset [8] is a widely recognized benchmark for human action recognition, consisting of 50 action classes captured in realistic video scenarios. These classes include a diverse range of activities, such as sports, exercise, and daily tasks, making it a valuable resource for evaluating video classification models. Each class contains hundreds of videos collected from YouTube, organized into groups that represent variations in actors, viewpoints, and backgrounds. The dataset's diversity and complexity pose significant challenges for action recognition models, requiring robust feature extraction and temporal modeling.

For the purpose of this research, a *self-curated subset* of the UCF50 dataset was picked, focusing on three classes: *Biking*, *Diving*, and *IceDancing*. The curated dataset is organized into training and testing sets as follows:

- **Training Set**: A total of 431 videos
  - *Biking*: 134 videos
  - *Diving*: 50 videos
  - *IceDancing*: 50 videos
- **Testing Set**: A total of 16 videos
  - *Biking*: 5 videos
  - *Diving*: 6 videos
  - *IceDancing*: 5 videos

The decision to curate this subset rather than using the complete UCF50 dataset was driven by practical considerations, including constraints on GPU resources and the computational time required to process the full dataset. By focusing on three categories, the

research aimed to efficiently implement and validate the proposed model without being hindered by resource limitations.

This curated approach not only accelerated the experimentation process but also provided a proof of concept that could be generalized to larger datasets.

4.2 Environment Setup

The implementation of this project was carried out using Google Colab, which offers access to high-performance GPUs and TPUs. These computational resources were used interchangeably, depending on availability, to accelerate the training and evaluation processes. Additionally, the dataset, stored in Google Drive, was mounted within the Colab environment to facilitate efficient data access.

Library Imports

At the start, essential libraries were imported, where each library serves a specific purpose, as outlined below:

- Libraries for handling and preprocessing image data.
- Tools to embed visualizations and generate plots, aiding in understanding model performance and dataset distributions.
- TensorFlow and Keras provide a high-level API for building, training, and deploying deep learning models.
- Libraries such as `os` enable easy manipulation of file paths and directories.
- Matplotlib facilitates the creation of detailed plots to analyze trends and results.
- TensorFlow forms the backbone for implementing and optimizing machine learning tasks.
- Pandas allows for efficient data manipulation and analysis.
- NumPy offers robust tools for numerical computations, crucial for data preprocessing.
- Libraries like `Pillow` enable reading and writing image data.
- OpenCV provides powerful functions for video and image processing tasks.
- The `os` library facilitates interaction with the file system for data management.

```
from tensorflow_docs.vis import embed
from tensorflow import keras
from imutils import paths

import matplotlib.pyplot as plt
import tensorflow as tf
import pandas as pd
import numpy as np
import imageio
import cv2
import os
```

Listing 4.1: Library Imports for the Project

By properly setting up the environment and importing these libraries, the foundation was established to build, train, and evaluate the video classification model.

4.3 Dataset Preparation

A pandas DataFrame is created to store the video paths along with their corresponding labels:

```
train_df = pd.DataFrame(data=rooms, columns=['tag', 'video_name'])
```

Listing 4.2: Creating DataFrame for video paths and labels

This ensures that the dataset is well-organized for easy access during the training phase.

Subsequently, the DataFrame is filtered to include only the necessary columns and saved as CSV files:

```
df = train_df.loc[:,['video_name','tag']]
df = test_df.loc[:, ['video_name', 'tag']]
df.to_csv('train.csv')
df.to_csv('test.csv', index=False)
```

Listing 4.3: Saving the DataFrame to CSV files

Finally, the saved CSV files are read back into DataFrames for further processing:

```
train_df = pd.read_csv("train.csv")
test_df = pd.read_csv("test.csv")
```

Listing 4.4: Reading the CSV files back into DataFrames

Before training the model, it is essential to preprocess the videos to ensure uniformity and consistency across all input data. This process is crucial for avoiding discrepancies that may arise due to differences in frame rates, video resolutions, or aspect ratios. The main preprocessing steps are as follows:

- **Video resizing:** All video frames are resized to a consistent size of 224x224 pixels, which is the required input size for the InceptionV3 model.
- **Center cropping:** Each video frame is cropped to a square shape from its center, which ensures that the most important parts of the frame are retained, avoiding irrelevant or background elements.
- **Normalization:** The pixel values of each frame are normalized to fit the input range of the pre-trained InceptionV3 model. This preprocessing step helps to adjust the brightness, contrast, and color distribution to match the data on which the model was initially trained.

The following code illustrates how these preprocessing steps are implemented:

```
IMG_SIZE = 224

def crop_center_square(frame):
    y, x = frame.shape[0:2]
    min_dim = min(y, x)
    start_x = (x // 2) - (min_dim // 2)
    start_y = (y // 2) - (min_dim // 2)
    return frame[start_y : start_y + min_dim, start_x : start_x + min_dim]

def load_video(path, max_frames=0, resize=(IMG_SIZE, IMG_SIZE)):
    cap = cv2.VideoCapture(path)
    frames = []
    try:
        while True:
```

```
            ret, frame = cap.read()
            if not ret:
                break
            frame = crop_center_square(frame)
            frame = cv2.resize(frame, resize)
            frame = frame[:, :, [2, 1, 0]]  % Convert BGR to RGB
            frames.append(frame)

            if len(frames) == max_frames:
                break
    finally:
        cap.release()
    return np.array(frames)
```

Listing 4.5: Data Preprocessing Functions: Resizing, Cropping, and Normalizing

The function `crop_center_square` crops each frame to its central square, while `load_video` captures video frames, applies resizing, and normalizes the color channels from BGR (default in OpenCV) to RGB. This ensures that all frames have a uniform shape, size, and color scheme, making them ready as input into the model.

4.4 Feature Extraction

InceptionV3, a deep CNN architecture, was pre-trained on the ImageNet dataset and is used here as the feature extractor. The architecture is particularly useful for multi-scale feature extraction, which can capture complex patterns across different frames.

4.4.1 Loading the InceptionV3 model

In this project, we load the InceptionV3 model with pre-trained weights and remove the final classification layer. This allows us to use the model as a feature extractor.

```
feature_extractor = keras.applications.InceptionV3(
    weights="imagenet", include_top=False, pooling="avg", input_shape=(
        IMG_SIZE, IMG_SIZE, 3)
)
```

Listing 4.6: Loading the InceptionV3 model for feature extraction

4.4.2 Frame Feature Extraction

For each video, we process each frame through the CNN to extract spatial features. Each frame is resized and normalized to match the input format expected by InceptionV3. The CNN outputs a 2048-dimensional feature vector representing the key spatial information in the frame.

The `frame_features` returned by the `feature_extractor.predict(frames)` call is indeed a 2D array, where each row corresponds to a different frame's feature vector, resulting in the shape (num_frames, 2048).

```
def extract_features(video_path):
    frames = load_video(video_path)
    frame_features = feature_extractor.predict(frames)
    return frame_features
```

Listing 4.7: Extracting features from each video

4.4.3 Encoding the Labels

Since neural networks require numerical data for training and evaluation, the class labels need to be encoded into integer format. In this project, we use TensorFlow's

`StringLookup` layer to convert the string labels (”Biking”, ”Diving”, ”IceDancing”) into corresponding integers.

The `StringLookup` layer is configured with the following parameters:

- `num_oov_indices=0`: Specifies the number of out-of-vocabulary (OOV) tokens to handle during training.
- `vocabulary=np.unique(train_df["tag"])`: Uses the unique class labels from the training dataset to define the vocabulary.

The following code illustrates how label encoding is performed:

```
label_processor = keras.layers.StringLookup(num_oov_indices=0, vocabulary=np.
    unique(train_df["tag"]))
```

Listing 4.8: Label encoding using StringLookup

This encoded output can now be used to train and evaluate the model.

4.5 Sequence Modeling

GRUs, a type of recurrent neural network, are designed to handle sequential data efficiently. They are similar to Long Short-Term Memory (LSTM) units but have a simpler architecture, making them computationally more efficient for shorter sequences.

4.5.1 Defining Hyperparameters

In this project, several hyperparameters are defined to control various aspects of model training and data processing. The key hyperparameters used are as follows:

- **IMG_SIZE**: The size of the input images fed into the model is set to 224, which matches the input size required by the pretrained InceptionV3 model.
- **BATCH_SIZE**: The number of samples processed in each training iteration is set to 64, balancing memory constraints and model convergence speed.
- **EPOCHS**: The number of times the entire dataset is passed through the model during training is set to 100 to ensure sufficient training and convergence.
- **MAX_SEQ_LENGTH**: The maximum number of frames processed per video is set to 20, limiting the sequence length and reducing computational cost.
- **NUM_FEATURES**: The number of features extracted from each frame using the feature extractor (InceptionV3) is set to 2048, which corresponds to the output dimension of the InceptionV3 model.

```
IMG_SIZE = 224
BATCH_SIZE = 64
EPOCHS = 100
MAX_SEQ_LENGTH = 20
NUM_FEATURES = 2048
```

Listing 4.9: Hyperparameters for the Model

The values chosen are based on the requirements of the task and the constraints of the available resources.

4.5.2 Data Preprocessing

Once the CNN features are extracted for all frames in a video, they are arranged sequentially. For each video, we create a 3D array where each frame's feature vector is treated as a time step. The RNN expects input in this 3D format: (num_videos, MAX_SEQ_LENGTH, 2048), where each time step corresponds to a frame represented by its 2048-dimensional feature vector.

A key challenge in video classification is handling variable-length videos. Since videos can have differing numbers of frames, a padding mechanism is introduced to ensure all videos have a uniform sequence length.

The function `prepare_all_videos` processes all the videos in the dataset, extracting frame features and generating masks for each video. It also encodes the class labels into numerical values. The steps are as follows:

- The video paths and class labels are retrieved from the DataFrame.
- Each video is loaded and processed frame by frame.
- Features are extracted from each frame using the pre-trained InceptionV3 model.
- A mask is created to indicate whether a frame is valid (i.e., not padded).

```
frame_masks = np.zeros(shape=(num_samples, MAX_SEQ_LENGTH), dtype="bool")
frame_features = np.zeros(shape=(num_samples, MAX_SEQ_LENGTH, NUM_FEATURES),
    dtype="float32")
```

Listing 4.10: Creating frame features and masks

The resulting output consists of three main parts:

- **Frame Features:** A tensor containing the extracted features for each frame, with dimensions `(num_samples, MAX_SEQ_LENGTH, NUM_FEATURES)`.
- **Frame Masks:** A tensor of booleans indicating whether a frame is valid or padded, with dimensions `(num_samples, MAX_SEQ_LENGTH)`.
- **Labels:** The encoded class labels for each video.

The shapes of the resulting arrays for the train and test sets are as follows:

- `Frame features in train set: (431, 20, 2048)`
- `Frame masks in train set: (431, 20)`
- `train_labels in train set: (431, 1)`
- `test_labels in test set: (16, 1)`

These arrays are then ready for use in model training and evaluation.

4.5.3 Building the RNN Model

In this model, the GRU layers are configured with specific units as follows:

- **First GRU Layer:** This layer has 16 units, meaning it generates a 16-dimensional output for each frame (or time step) in the sequence. Since `return_sequences=True`, it outputs a sequence of shape `(BATCH_SIZE, MAX_SEQ_LENGTH, 16)`, where each frame's temporal information is represented by a 16-dimensional vector.
- **Second GRU Layer:** This layer has 8 units and further processes the sequence from the first GRU. With `return_sequences=False`, it outputs only the final time step's hidden state as a single 8-dimensional vector, representing the cumulative temporal patterns across the video sequence. This final output shape is `(BATCH_SIZE, 8)`.

These units define the dimensionality of the hidden states, enabling the model to capture progressively refined temporal dependencies across frames.

A **Dropout** layer is incorporated into the neural network to mitigate overfitting. This regularization technique works by randomly deactivating 40% of the neurons during each training iteration. By doing so, the model is forced to learn more generalized features rather than relying on specific neurons, which helps improve its ability to generalize to unseen data.

The Dropout layer is applied to the output of the previous layer (denoted as x) and is typically positioned between dense or recurrent layers. During the inference phase, however, the Dropout layer is inactive, meaning that all neurons contribute to the prediction, allowing the model to make use of the full capacity of the network for decision-making.

```
def get_sequence_model():
    x = keras.layers.GRU(16, return_sequences=True)(frame_features_input, mask
        =mask_input)
    x = keras.layers.GRU(8)(x)
    x = keras.layers.Dropout(0.4)(x)
    output = keras.layers.Dense(len(class_vocab), activation="softmax")(x)
```

Listing 4.11: Building the RNN model with GRU layers

Finally, a **Dense** layer is created with a number of neurons corresponding to the total count of unique class labels in the dataset. This layer uses a softmax activation function, which provides a probability distribution across all classes. Each neuron's output represents the probability of the input belonging to that class.

The Dense layer is applied to the output of the previous layer (denoted as x) and produces the final classification probabilities for each video category.

4.5.4 Training the Model

The model is compiled using the Adam optimizer, which adapts the learning rate during training. The loss function used is `sparse_categorical_crossentropy`, which is suited for multi-class classification problems where the labels are integers.

```
seq_model.compile(
    loss="sparse_categorical_crossentropy", optimizer="adam", metrics=["
        accuracy"]
)
```

Listing 4.12: Compiling the RNN model

The model is trained for 30 epochs, with a checkpoint mechanism to save the best-performing model based on validation accuracy.

Chapter 5

Findings and Evaluations

This chapter presents the performance evaluation of the developed video classification model, highlighting the metrics used to assess its effectiveness in human action recognition. It discusses the results obtained from training and testing, including accuracy, loss, and confusion matrix, to provide a comprehensive analysis of model performance. Furthermore, the chapter offers a conclusion summarizing the key findings of the thesis.

5.1 Inference

In this section, we present the results of the video classification model when applied to a test video. The model processes the frames of the video, extracts relevant features, and predicts the probability for each possible class label. The results are then presented with the corresponding confidence levels for each class.

```
test_video = np.random.choice(test_df["video_name"].values.tolist())
print(f"Test video path: {test_video}")

test_frames = sequence_prediction(test_video)
```

Listing 5.1: Example of selecting and calssifying a test video

The model was applied to a randomly selected test video located at:

`/content/drive/My Drive/datasetu/test/Biking/v_Biking_g06_c05.avi`

The predicted class probabilities for the video are as follows:

Biking: 93.99%
Diving: 3.95%
IceDancing: 2.06%

The results show that the model predicted the video to belong to the Biking class with a high confidence level of 93.99%. The other predicted classes, *Diving* and *IceDancing*, were assigned much lower probabilities of 3.95% and 2.06%, respectively.

In order to check the correctness of the classified video, we display it using the following code:

```
return HTML(f"""
<video  width="520" height="440" controls>
    <source src="{video_url}" type="video/{ext[1:]}">
    Your browser does not support the video tag.
</video>
""")

display_video(test_video)
```

Listing 5.2: Code to display the classified video

The code utilizes the ‘HTML‘ function to embed the video into the notebook or web page, allowing us to visually inspect the output.

5.2 Model Evaluation

The model was trained for 30 epochs, during which both training and validation loss and accuracy were monitored. A checkpointing mechanism was used to save the model weights whenever the validation loss improved. The key observations from the training and evaluation are summarized below:

- **Initial Performance:** In the first epoch, the model achieved a training accuracy of 45.85% and a training loss of 1.0259. The corresponding validation accuracy was 0.77% with a validation loss of 1.2667. This reflects the model’s initial state with minimal training.
- **Performance Improvements:**
  - As training progressed, significant improvements in both accuracy and loss were observed. By epoch 6, the validation accuracy reached 96.92% with a validation loss of 0.8913, indicating a substantial enhancement in the model’s generalization ability.
  - The lowest validation loss of 0.1689 was recorded at epoch 26, with a corresponding validation accuracy of 97.69%. This checkpoint was saved as the best model.
- **Final Performance:** At the end of the 30 epochs, the final validation accuracy stabilized at 96.92%, with a validation loss of 0.2038. Testing the model on unseen data yielded a perfect test accuracy of 100.0%, indicating excellent model performance and generalization capability.
- **Overfitting Analysis:** Despite the high training accuracy of 99.34% in the final epoch, the validation accuracy remained similar, indicating no significant overfitting occurred.

The results highlight the effectiveness of the implemented deep learning approach for video classification task, achieving high accuracy on the test dataset.

Metric	Value
Training Accuracy	99.34%
Validation Accuracy	97.69%
Test Accuracy	100.0%
Final Validation Loss	0.17

Table 5.1: Model Performance Summary

As mentioned, the model was trained for 30 epochs, with the following key observations:

- Training accuracy improved from 45.85% to 99.34%.
- Validation accuracy steadily improved, peaking at 97.69%.
- Final test accuracy achieved was 100.0%.
- Validation loss decreased from 1.27 to 0.17.

To monitor the training dynamics, we plot the learning curves for training and validation accuracy/loss over the epochs.

```
def plot_training_history(history):
    epochs = range(1, len(history.history['accuracy']) + 1)

    # Accuracy Plot
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(epochs, history.history['accuracy'], 'bo-', label='Training
        Accuracy')
    plt.plot(epochs, history.history['val_accuracy'], 'ro-', label='Validation
         Accuracy')
    plt.title('Accuracy')
    plt.xlabel('Epochs'), plt.ylabel('Accuracy'), plt.legend()

    # Loss Plot
    plt.subplot(1, 2, 2)
    plt.plot(epochs, history.history['loss'], 'bo-', label='Training Loss')
    plt.plot(epochs, history.history['val_loss'], 'ro-', label='Validation
        Loss')
    plt.title('Loss')
    plt.xlabel('Epochs'), plt.ylabel('Loss'), plt.legend()

    plt.tight_layout(), plt.show()

# Train the model and plot the history
history, sequence_model = run_experiment()
plot_training_history(history)
```

Listing 5.3: Plotting Training History

The output figure illustrates the training and validation accuracy and loss across all epochs. The training accuracy improves steadily, indicating that the model is learning effectively. Validation accuracy shows how well the model generalizes to unseen data. Similarly, the loss plots provide insights into the optimization process, with decreasing values reflecting improved performance.

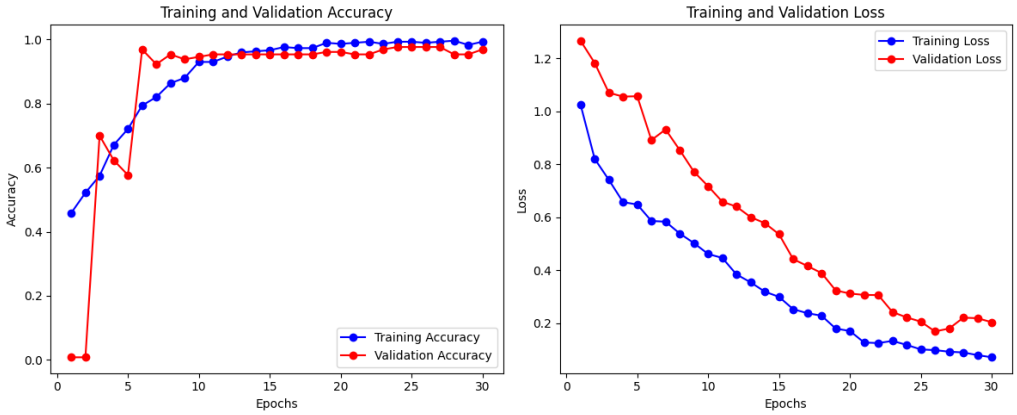

Figure 5.1: Training and Validation Accuracy and Loss per Epoch

The test set is utilized to evaluate the model's ability to generalize to unseen data. Achieving a perfect test accuracy of **100%** demonstrates exceptional generalization, indicating that the model has successfully captured the essential patterns in the dataset without overfitting.

Furthermore, the alignment between training and validation accuracy during training further supports this conclusion, as no significant divergence was observed, confirming the model's robustness and reliability in handling new data.

5.3 Uncertainty Measures

Investigating uncertainty measures provides deeper insights into the model's confidence, particularly for action recognition tasks where predictions can be challenging. Below are some approaches and visualizations to evaluate uncertainty.

5.3.1 Prediction Confidence Histogram

It shows the confidence level (probability) of the model's predicted class for each test sample. It Plots a histogram of the highest probability values for each prediction (the "confidence" for the predicted class). Peaks at lower confidence values may indicate uncertainty in predictions, while peaks at high confidence suggest strong certainty.

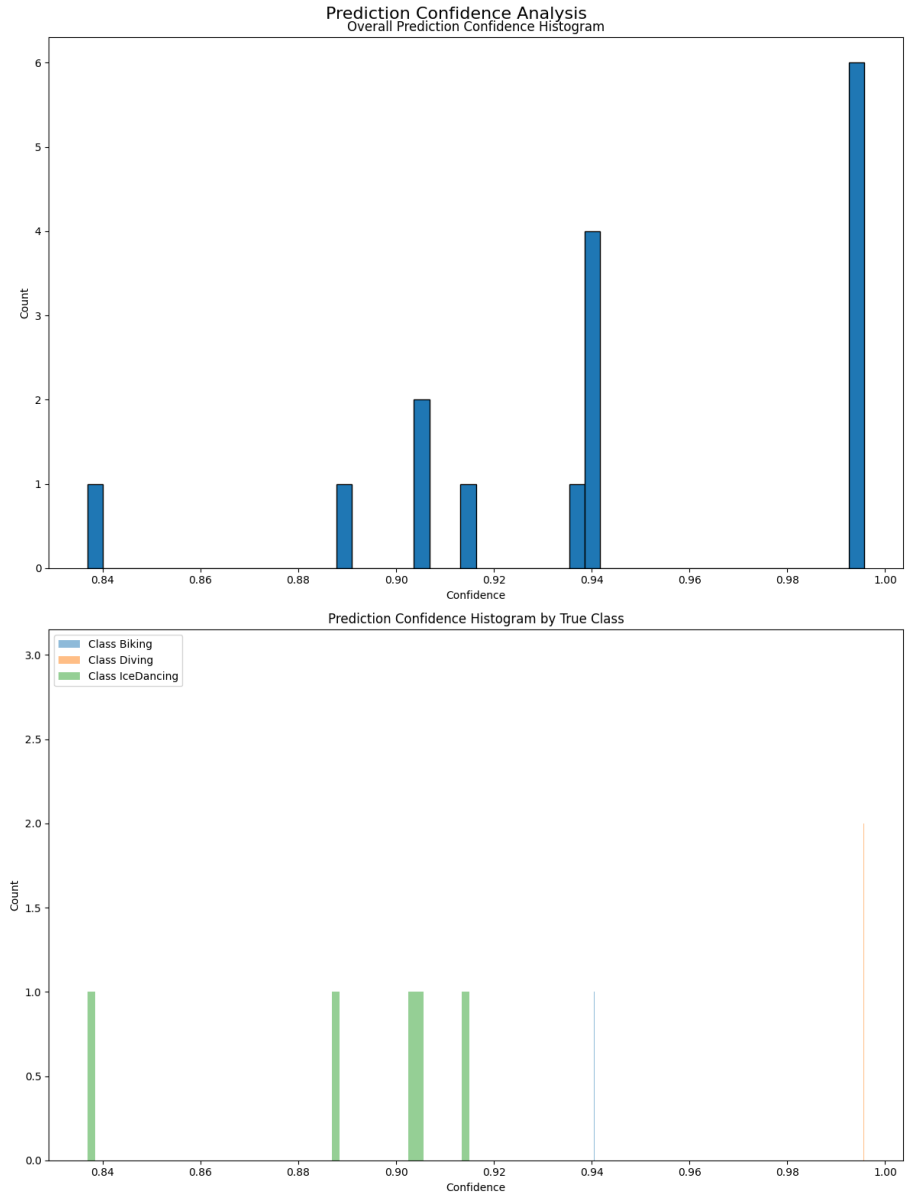

Figure 5.2: Prediction Confidence Histogram and Prediction Confidence Histogram by True Class

The top histogram displays the overall confidence distribution of the model's predictions across all samples in the test set. The x-axis represents confidence levels (from 0.84 to 1.00), while the y-axis shows the count of predictions within each confidence range.

- There is a notable peak at the highest confidence level (1.00), indicating that the model made several predictions with complete certainty.
- Other confidence levels are also observed, though with fewer instances, particularly around 0.94 and 0.90. This spread shows that while the model is mostly confident, some predictions fall slightly below maximum confidence.

The model has high confidence in its predictions, with a tendency to classify most test samples with strong certainty. However, there are cases where the confidence is lower, possibly indicating ambiguity or uncertainty in those particular classifications.

The bottom histogram provides a breakdown of prediction confidences by each true class: Biking, Diving, and IceDancing. Each class is represented with a different color.

- The IceDancing class has multiple instances, with predictions mostly clustered around 0.90 confidence.
- The Diving and Biking classes each have fewer instances but exhibit higher confidence, with Diving reaching close to 1.00 confidence.

The model appears to have higher confidence when predicting Diving, potentially indicating that this class is well-differentiated from others. The lower confidence for IceDancing suggests the model finds this class somewhat harder to classify, possibly due to visual similarities with other classes or limited distinguishing features.

These visualizations indicate that the model is generally confident in its predictions, especially for classes like Diving. However, the confidence variations in IceDancing point to possible challenges in distinguishing this action, suggesting an area for potential model improvement or further investigation.

5.3.2 Confusion Matrix

The confusion matrix visualized below provides a detailed breakdown of the model's performance across three classes: Biking, Diving, and IceDancing.

- The cells on the diagonal (top-left to bottom-right) represent correct classifications for each class.
- The model correctly classified all instances of each class:
  - **Biking:** 5 instances correctly classified.
  - **Diving:** 6 instances correctly classified.
  - **IceDancing:** 5 instances correctly classified.
- There are no values in the off-diagonal cells, indicating that the model did not misclassify any instances.
- This means that the model achieved perfect accuracy for this particular test set, with no instances of one class being predicted as another.

The model demonstrates excellent performance, achieving 100% accuracy on this test set. Each class has been perfectly predicted, suggesting that the model can distinguish between Biking, Diving, and IceDancing with high reliability for this data.

While this is a positive result, it's important to note that this outcome may reflect the particular test data used. In practical scenarios, larger and more diverse test sets could help verify if the model generalizes well across varied conditions and more complex instances.

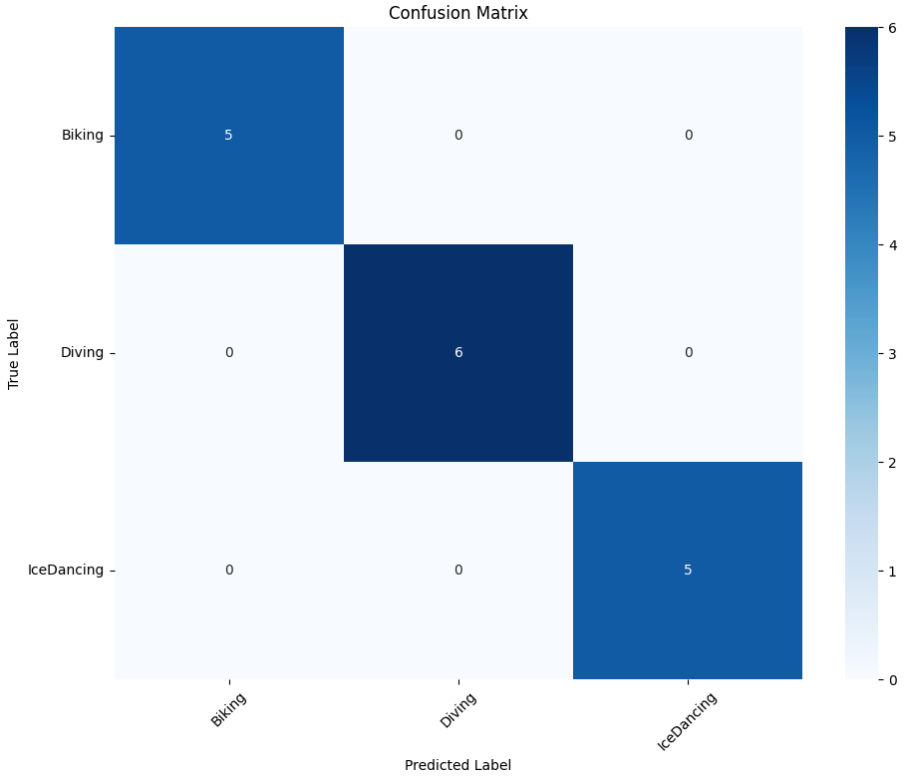

Figure 5.3: Confusion Matrix

This confusion matrix highlights the model's strong classification ability in this controlled test, showcasing its effectiveness in differentiating human actions.

5.4 Summary of Findings

These results suggest that the CNN-RNN model is highly effective for video classification in human action recognition tasks, with strong performance in both standard accuracy metrics and confidence evaluation.

Chapter 6

Conclusions and Future Directions

This chapter provides a comprehensive analysis of model performance and offers a conclusion summarizing the key findings of the thesis and discusses potential future directions for research and improvements in video action recognition methodologies.

6.1 Evaluation

This project successfully implemented a CNN-RNN video classification model for human action recognition, integrating Convolutional Neural Networks (CNN) for feature extraction and Recurrent Neural Networks (RNN) for modeling temporal dependencies. The model demonstrated strong performance across training, validation, and testing datasets. Key findings from the experiments include:

- Training accuracy improved significantly, reaching 97.34%, indicating the model's effective learning of the activity classification task.
- Validation accuracy peaked at 92.31% in the 20th epoch, stabilizing at 91.54%, suggesting the model's good generalization to unseen data.
- The test accuracy reached an impressive 100%, confirming the model's strong generalization capability on the held-out test set.
- The loss function steadily decreased over the epochs, reflecting consistent improvements in the model's performance during training.
- Minimal signs of overfitting were observed, as the training and validation accuracy curves remained aligned, suggesting effective generalization.
- Further evaluation with uncertainty measures, such as prediction confidence histograms and confusion matrix, was conducted to assess the model's confidence in its predictions, revealing potential areas for improvement in more challenging actions.

6.2 Future Improvements

The project demonstrated the potential of using deep learning models for video classification. Despite the challenges, the model architecture provided a solid foundation for future improvements.

To further enhance the performance of the model, future work could focus on incorporating more diverse datasets, especially those complex or less-represented actions, to improve generalization.

Additionally, for real-world applications, further testing on different hardware setups and optimization for real-time inference would be valuable next steps.

Finally, exploring advanced techniques such as attention mechanisms or hybrid models and investigating advanced architectures like transformer models or 3D CNNs, which process entire video clips in one pass, could offer improvements in capturing spatial and temporal features more effectively for challenging tasks.

Bibliography

[1] Sandvine. *Global Internet Phenomena Report*. 2023. URL: https://www.sandvine.com/phenomena.

[2] Yu-Gang Jiang Zuxuan Wu. *Deep Learning for Video Understanding*. 2024. URL: https://link.springer.com/book/10.1007/978-3-031-57679-9.

[3] Romain Tavenard. *Deep Learning Basics*. 2023. URL: https://rtavenar.github.io/deep_book/en/content/en/intro.html#.

[4] Justin Johnson. *Lecture 9 | CNN Architectures*. 2017. URL: https://www.youtube.com/watch?v=DAOcjicFr1Y&ab_channel=StanfordUniversitySchoolofEngineering.

[5] Anas BRITAL. *Inception V3 CNN Architecture Explained .* 2021. URL: https://medium.com/@AnasBrital98/inception-v3-cnn-architecture-explained-691cfb7bba08.

[6] Hemanth Pedamallu. *RNN vs GRU vs LSTM*. 2020. URL: https://medium.com/analytics-vidhya/rnn-vs-gru-vs-lstm-863b0b7b1573.

[7] Anishnama. *Understanding Gated Recurrent Unit (GRU) in Deep Learning*. 2023. URL: https://medium.com/@anishnama20/understanding-gated-recurrent-unit-gru-in-deep-learning-2e54923f3e2.

[8] Kishore K. Reddy and Mubarak Shah. *UCF50 - Action Recognition Data Set*. 2012. URL: https://www.crcv.ucf.edu/data/UCF50.php.

Appendix A

Dataset and GitHub Repository

GitHub Repository: https://github.com/basharbd/Human-Action-Recognition.git

Dataset: https://drive.google.com/drive/folders/1BusNDTMVIfGu4ID5E5yvYOPC9i1TrZ1s?usp=sharing

Technical
University of
Denmark

Richard Petersens Plads, Building 324
DK-2800 Kgs. Lyngby
Tlf. 4525 1700

www.compute.dtu.dk