UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Compression of Activation Signals from Partitioned Deep Neural Networks Exploring Temporal Correlation

Lucas Damasceno Silva

DM: 29/24

UFPA / ITEC / PPGEE
Campus Universitário do Guamá
Belém-Pará-Brasil
2024

UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Lucas Damasceno Silva

Compression of Activation Signals from Partitioned Deep Neural Networks Exploring Temporal Correlation

DM: 29/24

UFPA / ITEC / PPGEE

Campus Universitário do Guamá

Belém-Pará-Brasil

2024

UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Lucas Damasceno Silva

Compression of Activation Signals from Partitioned Deep Neural Networks Exploring Temporal Correlation

Submitted to the examination committee in the graduate department of Electrical Engineering at the Federal University of Pará in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering with emphasis in Telecommunications.

UFPA / ITEC / PPGEE
Campus Universitário do Guamá
Belém-Pará-Brasil
2024

Dados Internacionais de Catalogação na Publicação (CIP) de acordo com ISBD
Sistema de Bibliotecas da Universidade Federal do Pará
Gerada automaticamente pelo módulo Ficat, mediante os dados fornecidos pelo(a) autor(a)

S586c Silva, Lucas Damasceno.
Compression of Activation Signals from Partitioned Deep Neural Networks Exploring Temporal Correlation / Lucas Damasceno Silva. — 2024.
xv, 82 f. : il. color.

Orientador(a): Prof. Dr. Aldebaro Barreto da Rocha Klautau Júnior
Dissertação (Mestrado) - Universidade Federal do Pará, Instituto de Tecnologia, Programa de Pós-Graduação em Engenharia Elétrica, Belém, 2024.

1. Compressão. 2. DNN. 3. Sinais de ativação. 4. Correlação temporal. 5. Predição. I. Título.

CDD 621.3822

UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

"COMPRESSION OF ACTIVATION SIGNALS FROM PARTITIONED DEEP NEURAL NETWORKS EXPLORING TEMPORAL CORRELATION"

AUTOR: **LUCAS DAMASCENO SILVA**

DISSERTAÇÃO DE MESTRADO SUBMETIDA À BANCA EXAMINADORA APROVADA PELO COLEGIADO DO PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA, SENDO JULGADA ADEQUADA PARA A OBTENÇÃO DO GRAU DE MESTRE EM ENGENHARIA ELÉTRICA NA ÁREA DE TELECOMUNICAÇÕES.

APROVADA EM: 27/11/2024

BANCA EXAMINADORA:

____

Prof. Dr. Aldebaro Barreto da Rocha Klautau Júnior
(Orientador – PPGEE/ITEC/UFPA)

____

Prof. Dr. Leonardo Lira Ramalho
(Avaliador Interno – PPGEE/ITEC/UFPA)

____

Prof. Dr. Diego de Azevedo Gomes
(Avaliador Externo – UNIFESSPA)

VISTO:

____

Prof. Dr. Diego Lisboa Cardoso
(Coordenador do PPGEE/ITEC/UFPA)

Acknowledgments

First and foremost, I thank God, as I can affirm that the Lord has sustained me thus far. I bless His name, for He has guided my steps along this path, allowed me to experience these opportunities, and present the work I have developed up to this point. In His infinite goodness, God has surrounded me with people who share similar ambitions, and more than that, blessed me with friends who have been by my side until now.

I am grateful to my parents, whose love and encouragement have allowed me to persevere without looking back. Through their help and support, I have been able to aspire to ever-greater heights, and with their teachings, I have become who I am today. I thank my sisters, who stood by me, for their support, affection, partnership, and camaraderie over the years. I am grateful to my girlfriend, who made this journey lighter and gave me the strength to face the challenges and obstacles along the way.

I am grateful for the friendships I made at LASSE, for the moments of relaxation, learning, and collaboration. In particular, I thank Ailton Oliveira, who stood side by side with me through all the challenges faced, and Eduardo Guedes for his immense contribution to the completion of this work.

I thank my advisor, Aldebaro Klautau, for believing in my abilities and placing trust and credibility in my work. Thank you for all the support, explanations, teachings, and knowledge shared, which greatly contributed to my professional growth. I also thank the committee members, Leonardo Ramalho and Diego Gomes, for sharing their experiences, insights, and comments, which were invaluable for the final version of this work.

Lucas Damasceno

November 2024

Glossary

AF	Activation Functions. 11, 12
AI	Artificial Intelligence. 1
AP	Average Precision. 50
BDM	Block Distortion Measure. 20
BMA	Block Matching Algorithm. 19
CNN	Convolutional Neural Networks. 4–6, 9
DNN	Deep Neural Networks. 1, 4, 10
ES	Exhaustive Search. 20
FPN	Feature Pyramid Network. 40
FS	Full Search. 20
IoT	Internet Of Things. 1
IoU	Intersection Over Union. 50
MAD	Mean Absolute Difference. 20
mAP	Mean Average Precision. x, 50, 52–60
MBs	MacroBlocks. 19
MCP	Motion-Compensated Prediction. 19, 20
MSE	Mean Squared Error. 16, 20, 44–48
MV	Motion Vector. 19, 20, 29–32

PSNR Peak Signal-to-Noise Ratio. 20

QAT Quantization-Aware Training. 42, 43

ReLU Rectified Linear Unit. 13

RLE Run-Length Encoding. 25

RPE Residual Prediction Error. 20

RPN Region Proposal Network. 34, 35, 37, 44

SAD Sum Of Absolute Differences. 20–22

SNR Signal-to-Noise Ratio. 16, 17

SQ Scalar Quantization. 15

SSIM Structural Similarity Index. 44–48

TSS Three Step Search. 21, 22, 30, 51

List of Figures

2.1 Generic architecture of a fully-connected DNN. 4
2.2 Simplified architecture of a CNN. 5
2.3 Simplified architecture of a convolutional layer. 7
2.4 Convolution process of a convolutional layer with 3×3 kernel and stride 1. . . 7
2.5 Matrix with stride set to 2. 8
2.6 Process of edge filling the output of a convolutional layer. 9
2.7 Max-pooling layer operation. 9
2.8 Average pooling layer operation. 10
2.9 Connection between the output of the convolutional layers and the fully connected layers. 10
2.10 Single-layer perceptron without activation function. 11
2.11 Single-layer perceptron with activation function. 11
2.12 Sigmoid Activation Function Curve. 12
2.13 Hyperbolic Tangent Activation Function Curve. 13
2.14 ReLU Activation Function Curve. 14
2.15 Relationship between input and output of an uniform quantizer. 15
2.16 Relationship between input and output of a non-uniform quantizer. 17
2.17 General model for a non-uniform quantizer. 18
2.18 General block matching motion estimation. 19
2.19 Example path for convergence of TSS. 22
2.20 Huffman tree construction process. 24
2.21 Final Huffman Coding Model. 24
2.22 Run-Length Encoding process. 25

3.1 Overview of the proposed system. 26

3.2 Encoding block of the proposed system. 27

3.3 Decoding block of the proposed system. 30

3.4 Encoding block of the proposed bi-directional prediction. 31

3.5 Decoding block of the proposed bi-directional prediction. 32

4.1 Architecture of the Faster R-CNN with ResNet-50 Backbone. 33

4.2 Architecture of the Faster R-CNN with MobileNetV3 Backbone. 36

4.3 Architecture of the RetinaNet with ResNet-50 Backbone. 39

4.4 Comparison between frames at index $t-1$ and t. 45

4.5 Residual between frames. 45

4.6 Comparison between feature maps at first partition point. 46

4.7 Residual between feature maps at first partition point. 46

4.8 Comparison between feature maps at second partition point. 47

4.9 Residual between feature maps at second partition point. 47

4.10 Comparison between feature maps at third partition point. 47

4.11 Residual between feature maps at third partition point. 48

4.12 Comparison between feature maps at fourth partition point. 48

4.13 Residual between feature maps at fourth partition point. 49

4.14 Data rate at the first partition point. 52

4.15 Mean Average Precision (mAP) at the first partition point. 53

4.16 Data rate at the second partition point. 54

4.17 mAP at the second partition point. 55

4.18 Data rate at the third partition point. 56

4.19 mAP at the third partition point. 56

4.20 Data rate at the last partition point. 58

4.21 mAP at the last partition point. 58

List of Tables

2.1 Probability of occurrence of symbols. 23
2.2 Output codeword for each symbol. 25

4.1 Total number of experiments for each video sample. 49
4.2 Parameter values for Sample 1 using Faster R-CNN with a ResNet-50 backbone. 51

Contents

Acknowledgment **vi**

Glossary **vii**

List of Figures **ix**

List of Tables **xi**

Contents **xii**

Abstract **xiv**

Resumo **xv**

1 Introduction **1**

1.1 Contributions 3

2 Theoretical Foundations **4**

2.1 Convolutional Neural Networks 4

2.1.1 Convolutional Layer 6

2.1.2 Pooling Layer 9

2.1.3 Fully Connected Layer 10

2.1.4 Activation Functions 11

2.2 Quantization 14

2.3 Motion Estimation 18

2.3.1 Block Matching Algorithms 19

2.4 Huffman Coding 22

2.5 Run-Length Encoding 25

3 Proposed Method **26**

3.1 System Description 26

4 Experimentals and Results **33**

4.1 Network Models 33

4.1.1 Faster R-CNN with ResNet-50 Backbone 33

4.1.2 Faster R-CNN with MobileNetV3 Backbone 36

4.1.3 RetinaNet with ResNet-50 Backbone 39

4.1.4 Quantization of Models 41

4.2 Dataset and Correlation Analysis 44

4.3 Results 49

5 Conclusion and Future Work **59**

Bibliography **61**

Abstract

The use of artificial neural networks for object detection, along with advancements in 6G and IoT research, plays an important role in applications such as drone-based monitoring of structures, search and rescue operations, and deployment on hardware platforms like FPGAs. However, a key challenge in implementing these networks on such hardware is the need to economize computational resources. Despite substantial advances in computational capacity, implementing devices with ample resources remains challenging. As a solution, techniques for partitioning and compressing neural networks, as well as compressing activation signals (or feature maps), have been developed. This work proposes a system that partitions neural network models for object detection in videos, allocating part of the network to an end device and the remainder to a cloud server. The system also compresses the feature maps generated by the last layers on the end device by exploiting temporal correlation, enabling a predictive compression scheme. This approach allows neural networks to be embedded in low-power devices while respecting the computational limits of the device, the transmission rate constraints of the communication channel between the device and server, and the network's accuracy requirements. Experiments conducted on pre-trained neural network models show that the proposed system can significantly reduce the amount of data to be stored or transmitted by leveraging temporal correlation, facilitating the deployment of these networks on devices with limited computational power.

***Keywords* —** Compression, DNN, activation signals, temporal correlation, prediction

Resumo

O uso de redes neurais artificiais para detecção de objetos, juntamente com avanços na pesquisa de 6G e IoT, desempenha um papel importante em aplicações como monitoramento de estruturas por drones, operações de busca e resgate, e implantação em plataformas de hardware como FPGAs. No entanto, um desafio fundamental na implementação dessas redes em tais hardwares é a necessidade de economizar recursos computacionais. Apesar dos avanços substanciais na capacidade computacional, implementar dispositivos com recursos amplos continua sendo um desafio. Como solução, técnicas de particionamento e compressão de redes neurais, bem como compressão de sinais de ativação (ou *feature maps*), foram desenvolvidas. Este trabalho propõe um sistema que particiona modelos de redes neurais para detecção de objetos em vídeos, alocando parte da rede em um *end device* e o restante em um servidor na nuvem. O sistema também comprime os mapas de características gerados pelas últimas camadas no dispositivo final, explorando a correlação temporal, o que possibilita um esquema de compressão preditiva. Essa abordagem permite que redes neurais sejam incorporadas em dispositivos de baixo consumo de energia, respeitando os limites computacionais do dispositivo, as restrições de taxa de transmissão do canal de comunicação entre o dispositivo e o servidor, e os requisitos de precisão da rede. Experimentos conduzidos em modelos de redes neurais pré-treinadas mostram que o sistema proposto pode reduzir significativamente a quantidade de dados a serem armazenados ou transmitidos ao explorar a correlação temporal, facilitando a implantação dessas redes em dispositivos com poder computacional limitado.

***Palavras-chave* —** Compressão, DNN, sinais de ativação, correlação temporal, predição

Chapter 1

Introduction

The emergence of Deep Neural Networks (DNN) has significantly advanced research in various fields, including computer vision, speech recognition, audio recognition, and natural language processing. The rapid growth and apparent superiority of deep learning over traditional machine learning methods, combined with the evolution of computational resources, have enabled the widespread use of these networks in numerous applications. Examples include object detection, Big Data processing, and the continuous evolution of Artificial Intelligence (AI). This progress has opened doors for increasingly complex, precise, and efficient solutions across industries, driving innovation and enabling more sophisticated technologies in automation, predictive analysis, and human-machine interaction [1].

The growing demand for advanced services, coupled with the rapid increase in Internet of Things (IoT) devices and other connected technologies, necessitates that modern wireless communication systems expand to meet user expectations for quality of service, throughput, latency, connectivity, and security. Emerging standards like 5G, 6G, and beyond (xG) are poised to introduce transformative changes to wireless communication networks, enabling a fully connected environment that supports ubiquitous connectivity across a vast array of IoT devices. These advancements will facilitate seamless, reliable connections for billions of interconnected devices, driving a hyper-connected world where IoT can unlock new levels of automation, real-time monitoring, and data-driven decision-making across industries [2].

Thus, the use of DNNs, alongside advancements in wireless communication and IoT research, enables exploration of a wide range of use cases, along with their implications and challenges. The application of artificial neural networks for object detection plays a crucial role in emerging use-case scenarios, particularly in contexts such as drones. These networks

can be utilized in drones for search and rescue missions, structural monitoring, and agricultural applications, demonstrating their potential in fields that benefit from real-time data analysis, automation, and enhanced situational awareness [3].

However, a key challenge in implementing these networks on resource-constrained hardware is the need to economize computational resources. Despite substantial advances in computational capacity, deploying devices with ample resources remains challenging. To address this, techniques for partitioning and compressing neural networks have been developed [4] [5] [6], aiming to optimize DNN deployment on limited-resource hardware. These methods often employ compression techniques such as quantization and pruning, which reduce computational demands while maintaining model accuracy.

Most existing studies in the literature focus on compressing activation signals (also known as feature maps), which are the outputs of convolutional layers after the activation function, to achieve a compressed model. However, these studies primarily explore spatial redundancy in feature maps, as they are typically applied to image classification or object detection tasks in images [7] [8].

This work proposes a system that partitions neural network models for object detection in videos, allocating part of the network to an end device and the remainder to a cloud server. The system also compresses the feature maps generated by the last layers on the end device by exploiting temporal correlation, enabling a predictive compression scheme. This approach allows neural networks to be embedded in low-power devices while respecting the computational limits of the device, the transmission rate constraints of the communication channel between the device and server, and the network's accuracy requirements.

The remainder of this manuscript is structured as follows: Chapter 2 introduces the fundamental concepts used throughout this work, discussing the general architecture of the convolutional neural network models and their functionality, as well as the main compression techniques used to explore spatial and temporal correlation. Chapter 3 details the proposed method and the techniques used in its development. Chapter 4 describes the specific architecture of the network models utilized, provides an overview of the datasets, analyzes the spatial and temporal correlations in feature maps, and presents the experiments conducted along with the results obtained. Finally, Chapter 5 presents a summary of the results present in this work and provides directions for future work.

1.1 Contributions

The contributions of this work are:

- Analyzing the existence of temporal correlation in feature maps across all layers of the network model when using video inputs. To the best of the author's knowledge, this is the first time in the literature that temporal correlation is used to compress feature maps.
- Implementing quantized network models for object detection using the PyTorch Quantization API, addressing the gap in existing codes and documentations, which currently supports quantization primarily for simple classification models, making it difficult to apply to more complex models.
- Proposing a novel compression scheme that predicts feature maps by leveraging temporal correlation.
- Enabling the implementation of a security layer for the information extracted from the layers deployed on the edge device, where instead of sending raw data, the compressed feature maps are transmitted over the network. This compression inherently adds a level of security, acting as a layer of cryptography for the data.

Chapter 2

Theoretical Foundations

2.1 Convolutional Neural Networks

DNN are computational processing systems inspired by the way biological nervous systems, such as the human brain, operate. They consist of a large number of interconnected computational nodes, known as neurons, that work together in a distributed manner to learn from input data and optimize their final output [1] [9].

The basic structure of a DNN can be modeled as shown in Figure 2.1. Initially, the input, typically a multidimensional data structure, is loaded into the input layer, which distributes it to the hidden layers. The hidden layers make decisions based on the outputs of previous layers and evaluate how stochastic changes affect the final outcome, thereby performing the learning process [10].

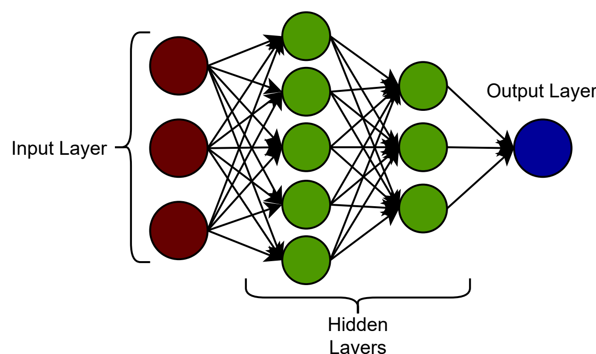

Figure 2.1: Generic architecture of a fully-connected DNN.

Convolutional Neural Networks (CNN) are a multi-layer deep learning architecture specifically designed to handle two-dimensional input data. Each layer of the network consists of

multiple two-dimensional planes, with each plane composed of several independent neurons. From the raw image vectors at the input to the final output, the entire network expresses a single perceptual scoring function, known as the weight [11].

CNNs are primarily used in pattern recognition applications involving images. They allow specific image features to be encoded into the architecture, making the network more suitable for image-focused tasks while reducing the number of parameters needed to configure the model.

A CNN is capable of successfully capturing the spatial and temporal dependencies in an image through the application of relevant filters. This architecture presents a better fit for image datasets by reducing the number of parameters and reusing weights, allowing the network to better understand the complexity of the image

One of the key differences in CNN architecture is the organization of neurons in three dimensions within the layers: the spatial dimensions of the input (height and width) and the depth. The role of a CNN is to reduce images to a format that is easier to process, without losing critical features necessary for accurate prediction, while ensuring the architecture is scalable to large datasets.

CNNs are composed of three types of layers: convolutional layers, pooling layers, and fully connected (dense) layers. When stacked, these layers form the architecture of a CNN. Figure 2.2 illustrates a simplified CNN architecture for classifying images from the MNIST (Modified National Institute of Standards and Technology) dataset.

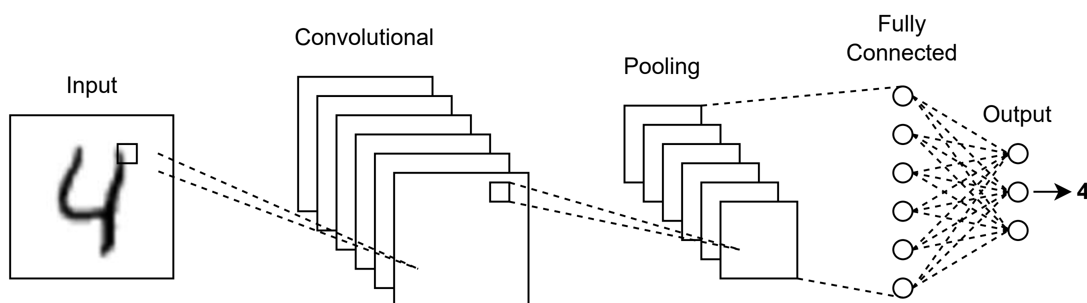

Figure 2.2: Simplified architecture of a CNN.

The basic functionality of the above example can be divided into five main areas:

1. The input layer receives the data to be processed by the rest of the network. For image applications, this typically involves storing the pixel values of the input image.

2. The convolutional layers, aided by activation functions like ReLU, are responsible for extracting features from the input images. These features may include simple patterns such as edges and textures in the initial layers, progressing to more complex patterns and object parts in deeper layers.

3. Pooling layers reduce the spatial dimensions (height and width) of the feature maps produced by the convolutional layers. This reduces computational complexity and the risk of overfitting, focusing on essential information (e.g., the presence of certain features).

4. After several convolutional and pooling layers, the high-dimensional feature maps are flattened into a one-dimensional vector. This vector is then passed to the fully connected layers, which act as classifiers using the features extracted and reduced by the earlier layers.

5. The final fully connected layer, typically using a softmax activation function, produces a probability distribution over the classes. In the case of the MNIST dataset, this means predicting the probability of each of the ten digits (0 through 9) based on the input image.

Using this method of transformation, CNNs are able to process the original input layer by layer through convolutional techniques and downsampling, ultimately producing class scores for classification and regression tasks. Next, we will present the individual layers in detail, along with their hyperparameters and connectivity.

2.1.1 Convolutional Layer

Convolutional layers play a crucial role in the functionality of CNNs. The parameters of these layers focus on the use of programmable kernels, which learn during training. These kernels are typically small in spatial dimensions but span the entire depth of the input. When data reaches a convolutional layer, each filter is convolved across the spatial dimensions of the input (i.e., the two-dimensional matrix of height and width) to produce a 2D activation map [12].

As the network performs convolutions over the input, the dot product is computed for each value in the kernel, as illustrated in Figure 2.3. The network learns kernels that activate when detecting specific features, commonly referred to as activations. Each kernel generates a corresponding feature (activation) map, which is stacked along the depth dimension to form the complete output volume of the convolutional layer [13].

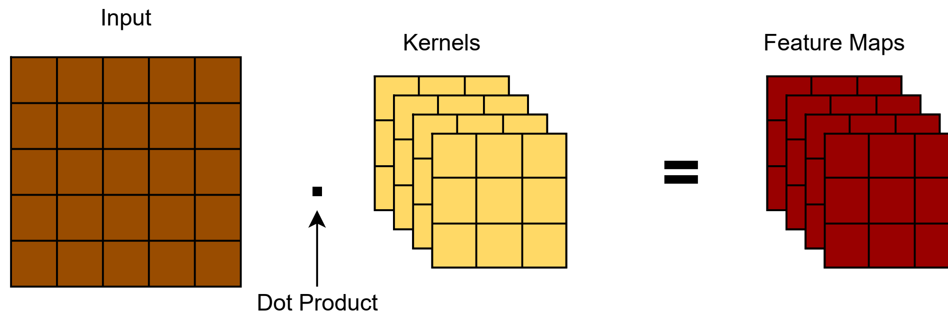

Figure 2.3: Simplified architecture of a convolutional layer.

Figure 2.4 illustrates a numerical example of how the convolution process in a convolutional layer operates. The two-dimensional input matrix (represented by the green matrix) is traversed in segments (represented by the orange matrices), with the step value defined by a hyperparameter called stride (which will be explained in more detail later). Each orange matrix is then combined with the kernel (represented by the yellow matrices) through the dot product, generating an activation map (represented by the red matrices). Throughout the training process, the network learns various kernels that produce multiple activation maps, which are then stacked to form the output of the convolutional layer.

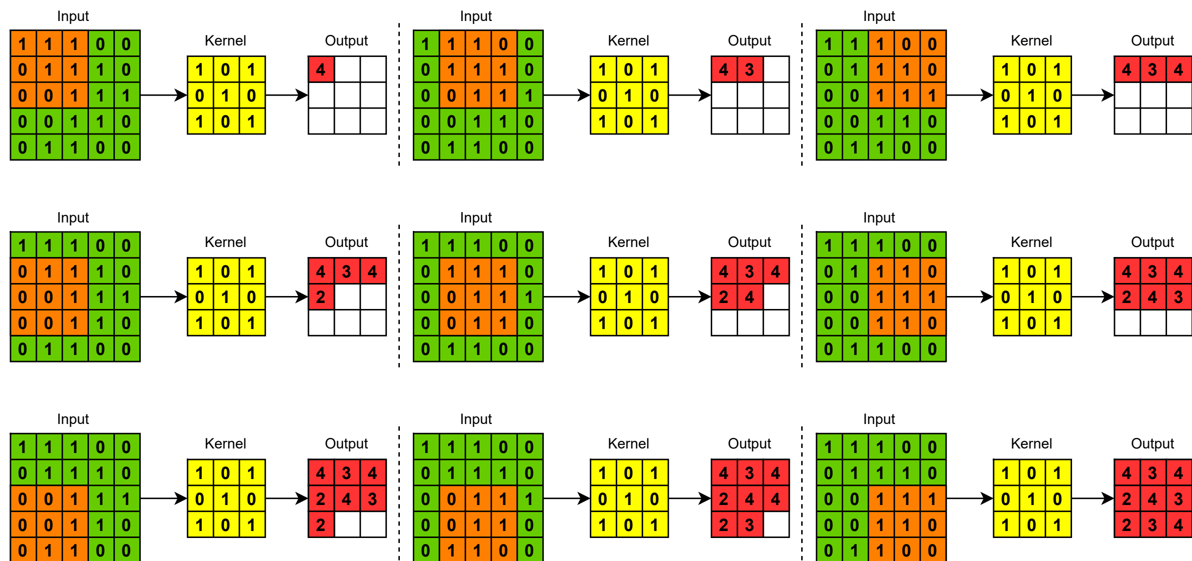

Figure 2.4: Convolution process of a convolutional layer with 3×3 kernel and stride 1.

As mentioned earlier, CNNs facilitate the creation of simpler models that scale better to large datasets. This is because each neuron in a convolutional layer is connected to only a small

region of the input volume, known as the receptive field. The dimensionality of this region is referred to as the size of the neuron's receptive field. For example, if we consider the input to the network as an image of size $64 \times 64 \times 3$ (an RGB image with a dimensionality of 64×64) and define the receptive field size as 6×6, each neuron in the convolutional layer will have a total of 108 weights (calculated as $6 \times 6 \times 3$, where 3 represents the depth of the input).

Convolutional layers can also significantly reduce model complexity by optimizing their output through three hyperparameters: depth, stride, and zero-padding. The depth of the output volume produced by convolutional layers can be defined by the number of neurons (or kernels) within the layer for a given region of the input. Reducing this hyperparameter can minimize the total number of neurons in the network but may also significantly decrease the model's pattern recognition capability.

Additionally, we can define the stride, which configures how the receptive fields position themselves around the spatial dimensions of the input. For instance, if we set the stride to 1, the receptive fields will overlap heavily, producing large activations. Conversely, increasing the stride reduces overlap and produces an output with lower spatial dimensions. Using the same input matrix from the example in Figure 2.4, we can set the stride to 2, as shown in Figure 2.5, thereby reducing the amount of overlap.

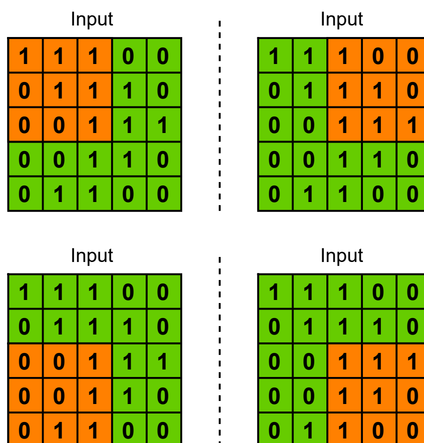

Figure 2.5: Matrix with stride set to 2.

Zero-padding is the process of adding zeros to the edges of the input, providing additional control over the dimensionality of the output volumes. This technique is exemplified in Figure 2.6, where padding is applied to the edges with zero values to preserve the dimensions of the input matrix.

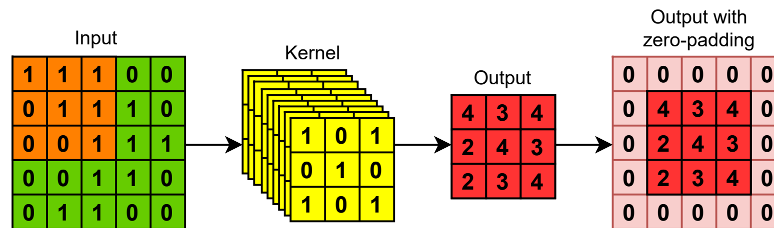

Figure 2.6: Process of edge filling the output of a convolutional layer.

2.1.2 Pooling Layer

Pooling layers aim to gradually reduce the dimensionality of the representation, thereby decreasing the number of parameters and the computational complexity of the model [14]. The pooling layer operates on each activation map and determines its dimension using two common methods: max-pooling and average pooling. Max-pooling layers perform an operation that selects the maximum element from the region of the activation map covered by the filter, helping to extract low-level features from the data, such as edges and points. An example of a max-pooling layer is illustrated in Figure 2.7.

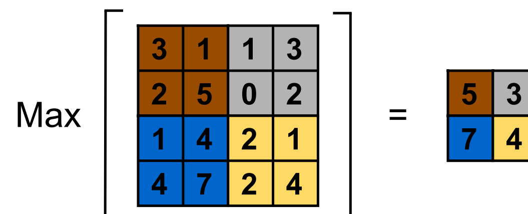

Figure 2.7: Max-pooling layer operation.

Average pooling layers operate by calculating the average of the elements in the activation map covered by the filter, as illustrated in Figure 2.8. In most CNNs, pooling layers use 2×2 filters applied with a stride value of 2 across the spatial dimensions of the input. This configuration reduces the activation map to 25% of its original size while maintaining the depth volume at its standard size.

Due to the destructive nature of pooling layers, two primary configurations are commonly observed. Typically, the stride and filter sizes are set to 2 and 2×2, respectively, allowing the layer to cover the entire spatial dimension of the input. Alternatively, overlapping pooling can be applied, where the stride remains 2, but the kernel size is increased to 3×3. For kernel sizes larger than 3×3, a significant decrease in model performance is generally observed.

$$\text{Avg}\left[\begin{array}{cccc} 3 & 1 & 1 & 3 \\ 2 & 5 & 0 & 2 \\ 1 & 4 & 2 & 1 \\ 4 & 7 & 2 & 4 \end{array}\right] = \begin{array}{cc} 2.75 & 1.5 \\ 4 & 2.25 \end{array}$$

Figure 2.8: Average pooling layer operation.

2.1.3 Fully Connected Layer

Fully connected layers, also known as dense layers, form the final part of the convolutional network and are responsible for learning the weights acquired in the convolutional layers, functioning similarly to the neurons in traditional DNNs. The activation matrices generated by the convolutional layers are reorganized into a one-dimensional vector through an intermediate layer called the flattening layer. The output of this flattening layer is then connected to every neuron in the first fully connected layer. Figure 10 illustrates how the convolutional and pooling layers connect to the fully connected layers.

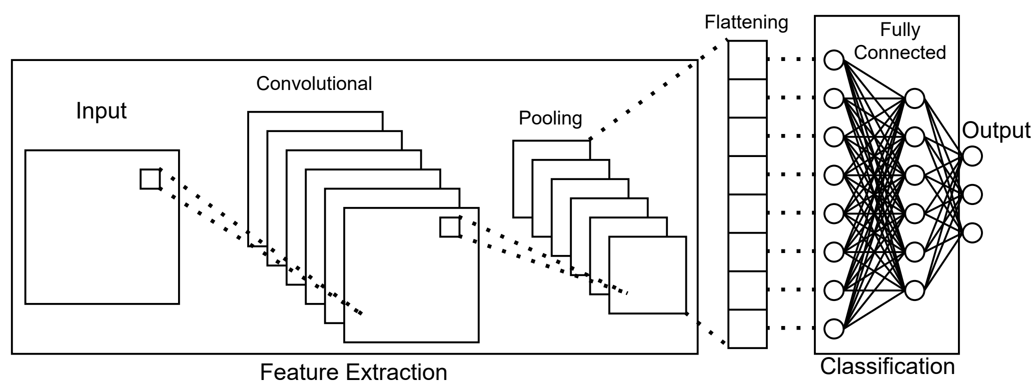

Figure 2.9: Connection between the output of the convolutional layers and the fully connected layers.

Since fully connected layers are densely connected, the majority of the network's parameters are concentrated in these layers. For example, consider two fully connected layers with approximately 4096 neurons each—the connection between these layers would require over 16 million weights. It is important to note that the structure of a fully connected layer is application-specific; for instance, a fully connected layer in a classification model functions differently from one in a segmentation model.

2.1.4 Activation Functions

Activation Functions (AF) in neural networks calculate the weighted sum of inputs and bias to determine whether a neuron should activate or not. AFs manipulate input data through a gradient-based process and generate an output for subsequent layers in the network. They can be either linear or nonlinear, playing a crucial role in controlling the outputs of the layers and enhancing the model's expressive capacity. This allows the network to better capture complex patterns, contributing to its ability to embody artificial intelligence [15].

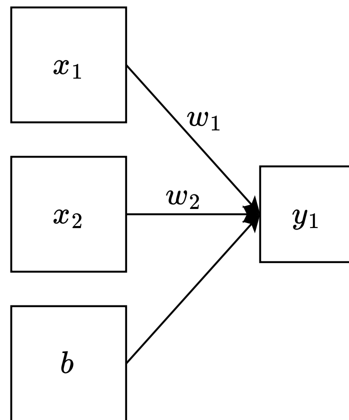

Figure 2.10: Single-layer perceptron without activation function.

In a single-layer perceptron, binary classification can be easily performed, as illustrated in Figure 2.10, where the model output y_1 is defined by:

$$y_1 = w_1 \times x_1 + w_2 \times x_2 + b. \tag{2.1}$$

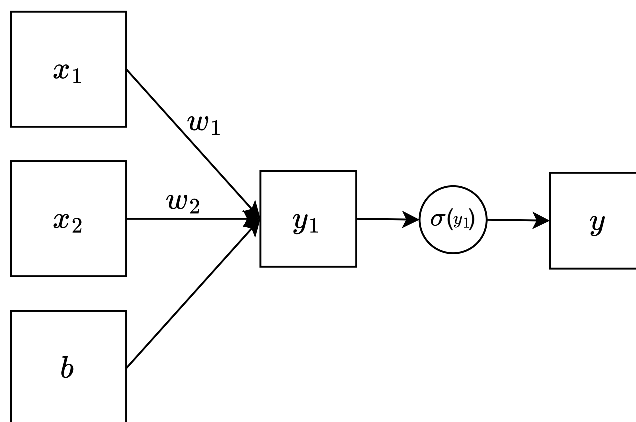

Figure 2.11: Single-layer perceptron with activation function.

However, because the core of the classifier is a linear equation, it cannot solve classification problems in nonlinear systems. To address this limitation, an AF is introduced into the perceptron, as shown in Figure 2.11. With this addition, the model output is defined as:

$$y = \sigma(w_1 \times x_1 + w_2 \times x_2 + b). \tag{2.2}$$

2.1.4.1 Sigmoid Function

The sigmoid function, also known as the logistic or squashing function, is a nonlinear function with a bounded output that was widely used as an activation function during the early years of deep learning research.

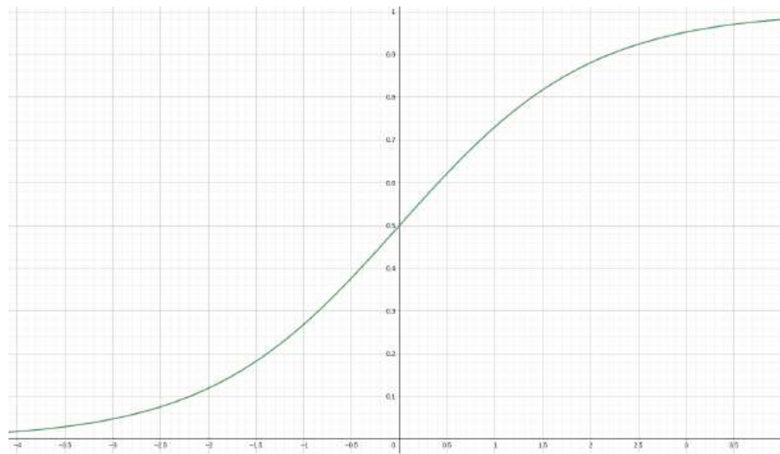

Figure 2.12: Sigmoid Activation Function Curve.

From the sigmoid function curve shown in Figure 2.12, we observe that the function exhibits a smooth saturability characteristic. Specifically, the slope of the graph approaches zero when the input is either very large or very small. When the slope is near zero, the gradient passed through the network diminishes, making it difficult to effectively train the network's parameters. Furthermore, the function's always-positive output limits the weight update direction to only one direction, which can slow the model's convergence rate. The sigmoid function is defined as:

$$y_1 = \frac{1}{1 + e^{-x}}. \tag{2.3}$$

2.1.4.2 Hyperbolic Tangent Function

The hyperbolic tangent function, commonly known as the tanh function, is an improved version of the sigmoid function, featuring symmetry and being centered at 0, as shown in Fig-

ure 2.13. The tanh function provides better training performance for multilayer neural networks compared to the sigmoid function. Its zero-centered output aids in the backpropagation process, making it more effective. The tanh function is defined as:

$$y_1 = \frac{1 - e^{-2x}}{1 + e^{-2x}}. \tag{2.4}$$

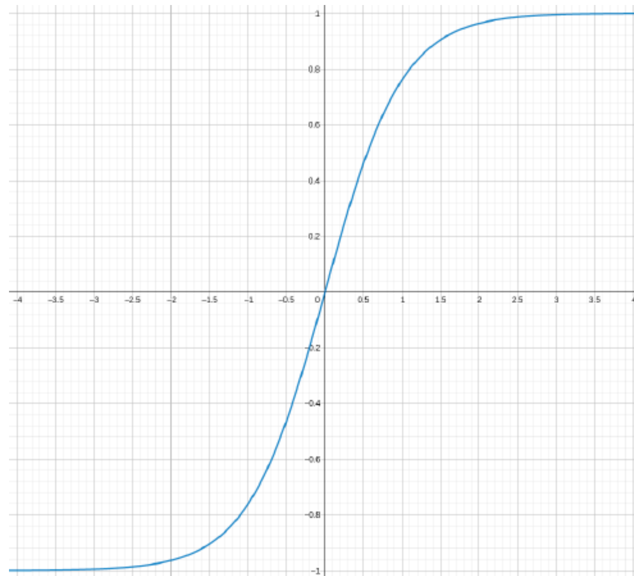

Figure 2.13: Hyperbolic Tangent Activation Function Curve.

2.1.4.3 ReLU Function

The Rectified Linear Unit (ReLU) function has become the most widely used activation function in deep learning applications. It offers faster learning and provides better performance and generalization compared to sigmoid and tanh functions. The ReLU function is nearly linear, preserving the linear properties of models and facilitating their optimization. Specifically, the ReLU activation function outputs 0 for any input values less than or equal to 0, as illustrated in Figure 2.14.

The ReLU activation function can be defined as:

$$y_1 = \begin{cases} 0; \text{ para } x \leq 0 \\ x; \text{ para} x > 0. \end{cases} \tag{2.5}$$

The main advantages of using the ReLU function are the reduction in computational cost, as it eliminates the need for exponential and division operations, and the introduction of sparsity in the data by setting negative values to zero.

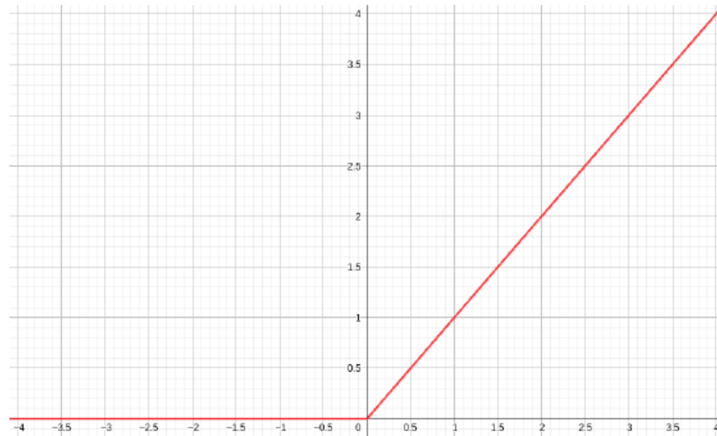

Figure 2.14: ReLU Activation Function Curve.

2.1.4.4 Softmax Function

The softmax function is another commonly used activation function in neural networks, particularly in the final layer of fully connected networks for classification tasks. It is designed to compute a probability distribution from a vector of real numbers, producing output values between 0 and 1, with the sum of the probabilities equal to 1. This makes it well-suited for multi-class classification problems, as the output can be interpreted as the probability of each class. The softmax function for a vector of K elements can be defined as:

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \text{ for i = 1,...,K and x = } (x_1,...,x_K). \tag{2.6}$$

2.2 Quantization

Quantization is a process of mapping a large set of values to a smaller set, which is inherently irreversible and lossy. Unlike sampling, quantization cannot fully reconstruct the original signal because the process discards certain information. However, the degradation caused by quantization can be relatively small and acceptable for certain applications, such as video and image compression in remote monitoring systems [16].

The main reasons for using quantization are: 1) Transmitting a digital signal with infinite amplitude values would require an extremely high transmission rate, which is impractical; 2) Human perception, in terms of vision and hearing, cannot easily distinguish between continuous signals with infinite amplitude levels and discrete signals with finite levels. This limitation allows for effective compression of audio and video signals, as our senses detect only finite

intensity differences.

Quantization is, therefore, a crucial part of compression algorithms for both audio and video data. There are two primary methods of quantization: scalar quantization, where individual samples are quantized one by one, mapping each input scalar value to an output scalar value and vector quantization, where samples are processed in blocks, allowing the quantization of a group of values simultaneously.

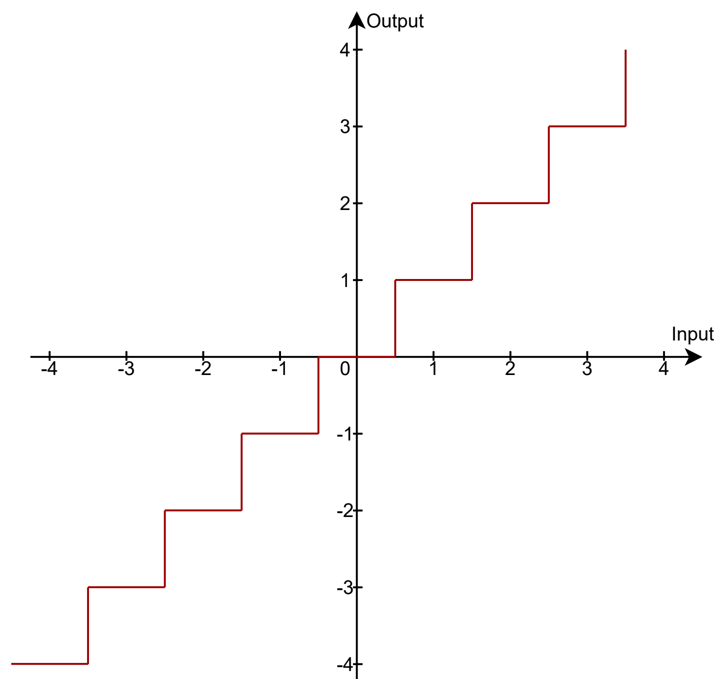

Figure 2.15: Relationship between input and output of an uniform quantizer.

Scalar Quantization (SQ) is a memoryless and instantaneous process, meaning the quantization of each sample is independent of other samples. It can be classified into two types: uniform and non-uniform quantization. The most common type is the uniform quantizer, which functions as a straightforward rounding mechanism. In this method, each sample is mapped to the nearest value from a finite set of quantization levels [17].

To better explain the uniform quantization process, assume that the amplitude of the input signal to the quantizer ranges between a maximum value $V > 0$ and a minimum value $-V$. Any amplitude of the non-quantized signal below $-V$ or above V is discarded. The range $[-V, V]$ defines the operational limit of the quantizer, and the error introduced by discarding values outside this range is known as overload distortion. Figure 2.15 illustrates the relationship between the input and output of an uniform quantizer.

The amplitude range $[-V, V]$ is divided into L quantization levels, each spaced apart by a value Δ, known as the step size. In a uniform quantizer, the step size is fixed across all quantization levels. For L quantization levels, the step size for a signal with amplitude $2V$ is given by:

$$\Delta = \frac{2V}{L}. \tag{2.7}$$

Within the amplitude range supported by the quantizer, the spacing between the continuous value and the discrete value is referred to as granularity, while the error introduced by this spacing is known as quantization noise or granular distortion. Both quantization error and overload distortion are inherent functions of the quantization process, and depend on the statistics of the input signal. It is important to balance these two sources of error.

Each quantization level at the output of the quantizer is mapped to a codeword of R bits, representing the quantized sample value. The number of quantization levels is expressed as:

$$L = 2^R = \frac{2V}{\Delta} \,. \tag{2.8}$$

For a uniform quantizer, the quantization error e is a random variable bounded by $-\frac{\Delta}{2} \leq e \leq \frac{\Delta}{2}$. When the step size Δ is sufficiently small, the quantization error can be considered uniformly distributed and uncorrelated with the input. Using Equation 2.8 and noting that the quantization error has a mean of 0, its variance (which equals the Mean Squared Error (MSE)) is given by:

$$\sigma_e^2 = \int_{-\frac{\Delta}{2}}^{\frac{\Delta}{2}} (\frac{1}{\Delta}) e^2 de = \frac{\Delta^2}{12} = \frac{V^2}{3L^2} = \frac{V^2}{3(2^{2R})} \,. \tag{2.9}$$

From this, we can observe that, considering the behavior of the uniform quantizer, as the number of quantization levels $L \to \infty$, or the step size $\Delta \to 0$, or the number of bits per quantized sample $R \to \infty$, the quantization error $e \to 0$.

The loading factor of a random input signal, with a mean of 0 and a standard deviation of σ_g, and a peak value of V, is defined by $\alpha = \frac{V}{\sigma_g}$. Noting that the signal has a mean of 0, the average power of the input signal is then defined as $P = \sigma_g^2$.

Thus, the Signal-to-Noise Ratio (SNR) at the output of a uniform quantizer with negligible saturation distortion is given by:

$$SNR = \frac{P}{\sigma_e^2} = (\frac{\sigma_g^2}{V^2}) 2^{2R} = (\frac{3}{\alpha^2}) 2^{2R} \,, \tag{2.10}$$

where it can be observed that the SNR increases exponentially as the number of bits per sample R increases. The SNR can be expressed in dB through the equation:

$$SNR = 6R - 20 log \alpha + 4.77 = 6R - 7.27 \text{ (dB)}, \tag{2.11}$$

where for each additional bit, an increase of 6 dB can be achieved for the SNR. Note that in the second equation, the value of α has been replaced with 4, which corresponds to a common choice for the loading factor.

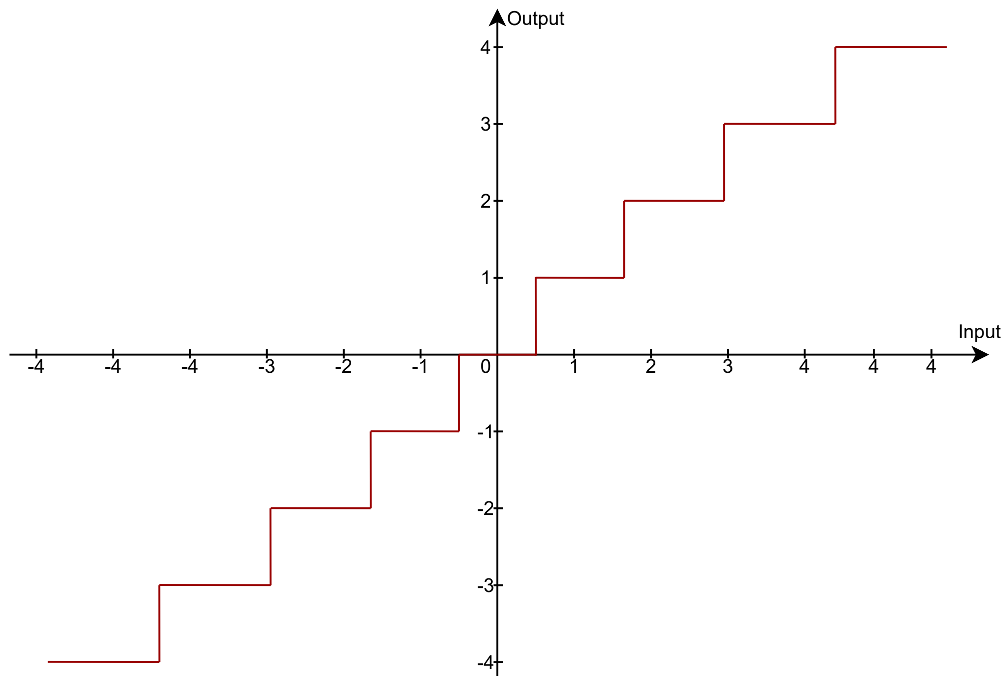

Figure 2.16: Relationship between input and output of a non-uniform quantizer.

The optimal quantization level in each quantization region should be chosen as the centroid (conditional expected value) of that region. However, since an ideal quantizer requires advanced knowledge of the signal's statistics and variations in power level, it has limited practical applications. In contrast, a uniform quantizer yields a higher (optimal) output SNR when the signal amplitude follows a uniform distribution over the dynamic range $[-V, V]$.

For sources with non-uniform distributions, an efficient approach is to employ a non-uniform quantizer that utilizes a variable step size. For instance, when dealing with an input that follows a Gaussian distribution with a mean of zero, one can design a quantizer with smaller spacings near zero and larger spacings at the extremes, as illustrated in Figure 2.16.

There are two main advantages to using a non-uniform spacing of quantization levels: 1) it is possible to significantly increase the dynamic range that can be accommodated for a given number of bits per sample; 2) it is possible to design quantizers tailored to the statistics of the input signal, thereby considerably increasing the SNR.

A general model for any non-uniform quantizer with a finite number of levels is shown

in Figure 2.17. The input x is first transformed through a non-linear compressor, which acts as a variable gain amplifier, attenuating large amplitudes and amplifying small amplitudes. The output $y = G(x)$ is then quantized with an uniform quantizer, producing an output $\hat{y}$, which is then transformed through a non-linear expander, in order to restore the signal samples to their original levels, thus generating an output signal $\hat{x} = G_{-1}(\hat{y})$. The combination of the compressor and the expander is referred to as a compander [17].

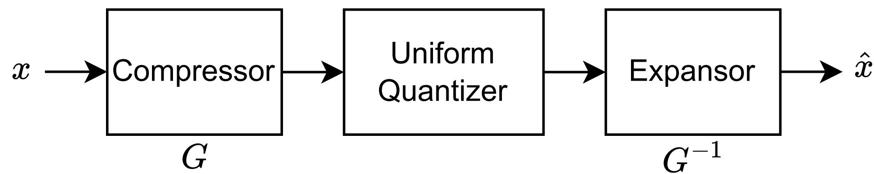

Figure 2.17: General model for a non-uniform quantizer.

2.3 Motion Estimation

Video compression is essential in multimedia technologies, as it allows for efficient storage and transmission of video content. Since video files are often large due to their detailed data, compressing them without significantly sacrificing quality is critical for streaming, broadcasting, and reducing storage requirements [18] [19].

The primary goal of video compression is to minimize file size by eliminating redundancies both within a single frame (intra-frame compression) and between consecutive frames (inter-frame compression). Two key types of redundancies are targeted by compression algorithms:

1. Spatial Redundancy: This refers to the redundancy within a single frame, where neighboring pixels often have similar color or intensity values. Compression techniques like JPEG exploit this redundancy by transforming the pixel data into a more compact form.

2. Temporal Redundancy: This refers to the redundancy between consecutive frames. For example, in many videos, much of the scene remains static from one frame to the next. Inter-frame compression techniques like motion estimation and motion compensation take advantage of this redundancy by identifying the motion of objects and predicting frames based on previous or future frames.

Motion estimation, in particular, plays a vital role in reducing temporal redundancy. Instead of transmitting entire frames, the encoder sends a Motion Vector (MV) that describes how blocks of pixels have moved between frames. This approach is fundamental to modern video codecs such as H.264, HEVC, and VP9. By accurately estimating and compensating for motion, these codecs achieve high compression ratios while maintaining excellent video quality [20].

2.3.1 Block Matching Algorithms

The Block Matching Algorithm (BMA) is a widely used technique for motion estimation. In this approach, the current frame is divided into non-overlapping MacroBlocks (MBs) of size $N \times M$. Each macroblock in the current frame is compared with a corresponding macroblock and its neighboring blocks in the reference frame. This process identifies displacement vectors that indicate how the macroblocks move from one position to another between frames [21].

For each macroblock in the current frame, the BMA searches for the best matching macroblock of the same size ($N \times M$) within a defined search area in the reference frame. The position of this matching macroblock determines the MV for the current macroblock, as shown in Figure 2.18. This MV has two components: horizontal and vertical, which can be either positive or negative. A positive value indicates motion to the right or downward, while a negative value indicates motion to the left or upward.

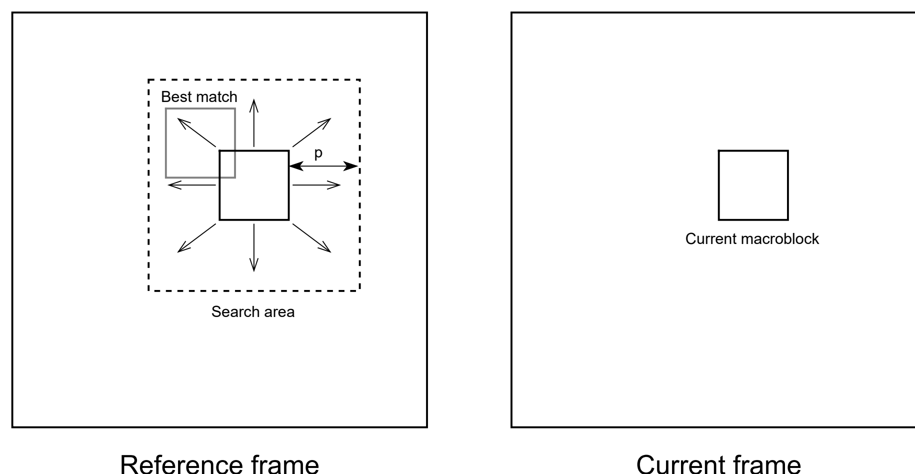

Figure 2.18: General block matching motion estimation.

These MVs are then used to generate the Motion-Compensated Prediction (MCP) for the current frame, using block motion compensation. The MVs are encoded through entropy

coding, and the Residual Prediction Error (RPE) between the current frame and the MCP is encoded using a combination of transform coding, quantization, and entropy coding.

At the decoder side, the MVs and RPE are decoded and used to reconstruct the MCP from the reference frame. The decoded RPE is then applied to the reconstructed MCP to recover the current frame.

The search area for finding a matching macroblock is typically limited to a region extending up to *p* pixels on all four sides of the corresponding macroblock in the reference frame, where *p* is the search parameter. Larger motions require a higher *p* value, increasing the computational demand.

The quality of the match is measured using a Block Distortion Measure (BDM), such as Mean Absolute Difference (MAD), Sum of Absolute Differences (SAD), or MSE. The macroblock that produces the lowest distortion cost is considered the best match for the current macroblock [22].

For a current macroblock *C* of size $N \times N$ and a candidate macroblock *R* in the reference frame, displaced by (v_x, v_y) the MAD, SAD, and MSE are calculated as follows:

$$MAD = \frac{1}{N \times N} \sum_{i=1}^{N} \sum_{j=1}^{N} |C(i,j) - R(i+v_x, j+v_y)| \ , \tag{2.12}$$

$$SAD = \sum_{i=1}^{N} \sum_{j=1}^{N} |C(i,j) - R(i+v_x, j+v_y)| \ , \tag{2.13}$$

$$MSE = \frac{1}{N \times N} \sum_{i=1}^{N} \sum_{j=1}^{N} (C(i,j) - R(i+v_x, j+v_y))^2 \ . \tag{2.14}$$

The simplest algorithm for motion estimation, known as Full Search (FS) or Exhaustive Search (ES), performs an exhaustive search for the best matching block within the defined search area. This involves moving the correlation window to every possible candidate position within the search area. As a result, FS identifies the most accurate match and achieves the highest Peak Signal-to-Noise Ratio (PSNR) compared to any other block matching algorithm. However, this approach comes with significant computational complexity.

To reduce and optimize this complexity, various fast block matching algorithms have been developed. If an algorithm can reduce computational effort while maintaining the same quality as FS, it is referred to as a lossless block matching algorithm. On the other hand, if the algorithm sacrifices some quality to improve speed, it is considered a lossy block matching algorithm [23] [24].

2.3.1.1 Three Step Search Block Matching Algorithm

The Three Step Search (TSS) Block Matching Algorithm is one of the most popular motion estimation algorithms used in video compression. It balances computational efficiency and accuracy, making it suitable for real-time applications like video streaming and conferencing.

The TSS algorithm finds the best match for a block in the reference frame through a hierarchical approach that progressively narrows down the search window [22]. The TSS algorithm divides the search process into three distinct steps:

- **Step 1: Initial Search Window**
  - Set Search Parameters: Assume the maximum displacement or search range is p. This defines how far the algorithm will search for a matching block. A typical value of p is 7 or 15.
  - Starting Point: The algorithm starts at the center of the search window in the reference frame (typically at the same position as the current block).
  - Initial Step Size: he initial step size is half the search range, following the equation $(p + 1)/2$.
  - Search 8 Neighbors: The algorithm evaluates the match at 8 points around the center, spaced by the step size in a diamond pattern.
  - Select the Best Match: The matching quality is evaluated for each of the 9 points (the 8 neighbors plus the center), usually using the SAD. The block with the lowest SAD is chosen as the new center for the next step.
- **Step 2: Reduce Step Size**
  - Halve the Step Size: The step size is reduced by half, from $(p + 1)/2$ to $(p + 1)/4$.
  - Search 8 Neighbors Again: The algorithm now searches around the new center in a similar diamond pattern with the smaller step size.
  - Select the Best Match: Again, the block with the lowest SAD from the 9 points becomes the new center for the final step.
- **Step 3: Final Search**
  - Further Reduce Step Size: The step size is reduced again to $(p + 1)/8$.

- Search the 8 Neighbors: A final search is conducted around the new center.
- Select the Final Match: The block with the lowest SAD is selected as the best match for the current block, and its corresponding displacement from the original position is the motion vector.

The general idea is shown in Figure 2.19, where the green points represent the first step, the blue points represent the second step, and the red points represent the final step. For a search range of $p = 7$, the full search method requires 225 points, while the three-step search method only needs $9 + 8 + 8 = 25$ points. Due to its simplicity and reasonable performance, the TSS is widely used for research purposes [25].

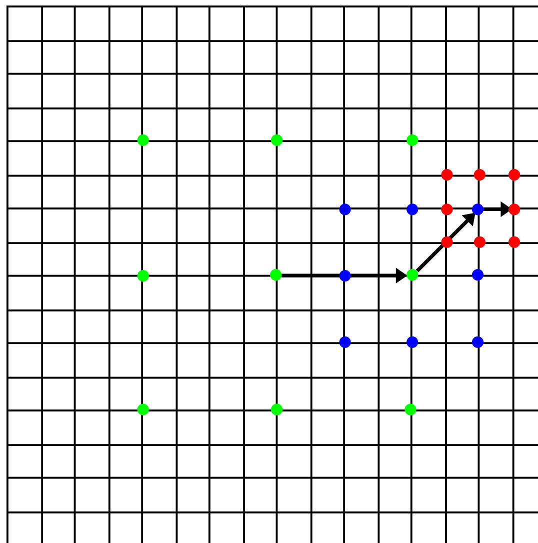

Figure 2.19: Example path for convergence of TSS.

2.4 Huffman Coding

Huffman coding is an entropy-based algorithm used for lossless data compression, where a variable-length code table is employed to encode input symbols. This table is derived from the estimated probability of occurrence of each symbol, with more likely symbols assigned shorter codewords, and less likely symbols assigned longer codewords. Today, Huffman coding is often used as a back-end for other compression techniques [26] [27].

In image compression, pixels are treated as symbols. Frequently occurring symbols are assigned fewer bits, while less frequent symbols are assigned more bits. The main steps in

Huffman coding are: 1) Construct a Huffman tree based on the input data; 2)Assign codes to symbols by traversing the Huffman tree. The process for constructing a Huffman tree is as follows:

1. Begin by creating a leaf node for each symbol and assigning it a priority based on its probability of occurrence.
2. Arrange the nodes in decreasing order of priority.
3. Choose the two nodes with the lowest priority (i.e., the lowest occurrence frequency).
4. Generate a new intermediate node, assigning it a priority equal to the sum of the priorities of the two selected nodes.
5. Repeat steps 3 and 4 until all nodes merge into a single tree.

Symbol	000	001	010	011	100	101	110	111
Probability	0.1	0.06	0.4	0.05	0	0.09	0.2	0.1

Table 2.1: Probability of occurrence of symbols.

For example, consider a set of pixels with the probability of occurrence for each symbol, as outlined in Table 2.1. The process of constructing the Huffman tree is illustrated in Figure 2.20. Once the tree is built, codes are assigned to each input symbol by traversing the tree from the root node (at the top) to each leaf node. At each step, assign a "0" to left branches and a "1" to right branches, as shown in Figure 2.21. The resulting codeword for each symbol and the average length after encoding are summarized in Table 2.2.

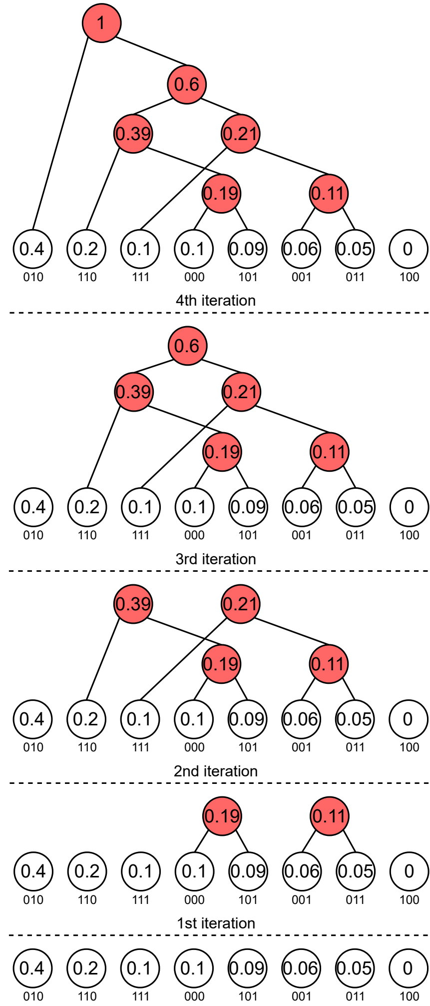

Figure 2.20: Huffman tree construction process.

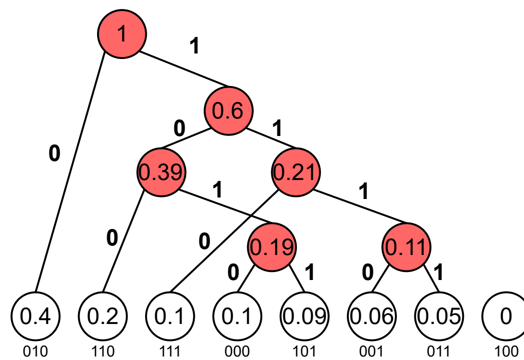

Figure 2.21: Final Huffman Coding Model.

Decoding is a straightforward process that only requires the receiver to have knowledge of the Huffman tree. The decoding begins by reading the encoded data and following the tree: a "0" directs the traversal to the left branch, and a "1" directs it to the right branch. Whenever a

leaf node is reached, the corresponding symbol stored in that leaf node is read and decoded.

Symbol	000	001	010	011	100	101	110	111
Codeword	1010	1110	0	1111		1011	100	110
Medium length								2.5

Table 2.2: Output codeword for each symbol.

2.5 Run-Length Encoding

Run-Length Encoding (RLE) is a lossless compression technique that is especially effective for data with repeated values. It works by replacing consecutive occurrences of a symbol with the count of its occurrences followed by the symbol itself [28]. RLE can be applied to various types of data, but its compression efficiency depends heavily on the data content. If the data lacks significant repetitions, RLE may actually increase the file size rather than reduce it.

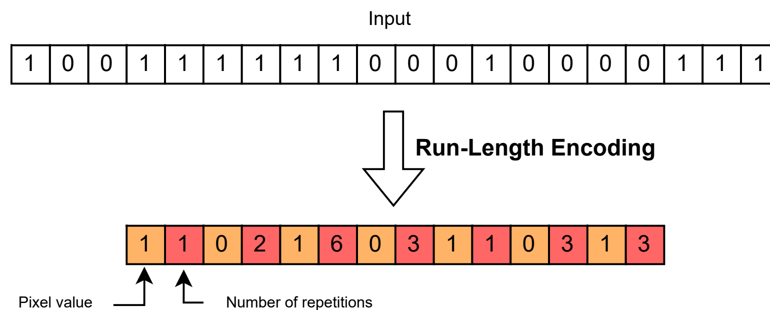

Figure 2.22: Run-Length Encoding process.

Although the ease of implementation and execution has made RLE an attractive alternative to more complex compression algorithms, its efficiency is maximized when the data consists of only two symbols in its bit pattern, with one symbol being more frequent than the other. In this case, RLE encodes the input data set into two bytes: the first byte represents the pixel intensity value, while the second byte indicates the number of consecutive pixels that share the same intensity, as illustrated in Figure 2.22.

Chapter 3

Proposed Method

3.1 System Description

This dissertation proposes a CNN model partitioning and compression system for object detection in videos, aimed at reducing the computational cost of the model. This makes it feasible to embed these networks into devices with computational limitations. Additionally, the work introduces a new compression methodology for activation maps by exploring their temporal correlation. This approach reduces the transmission rate needed to send data between parts of the network allocated on different devices, without significantly affecting the final model's classification performance.

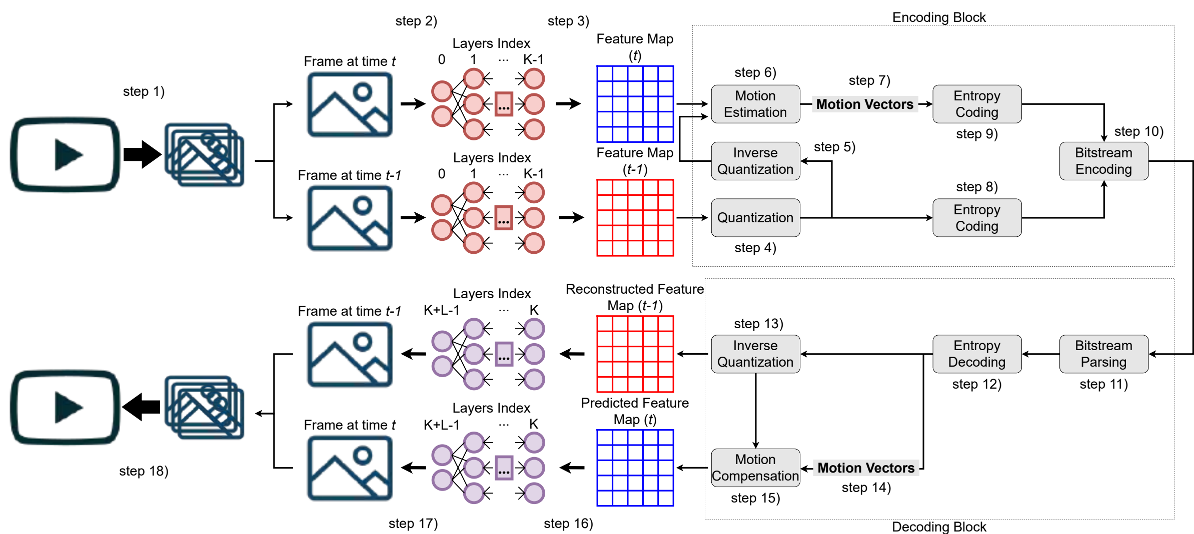

Figure 3.1: Overview of the proposed system.

Figure 3.1 presents an overview of the proposed system. Initially, the original network

model M, consisting of $K + L - 1$ layers, is quantized to 8-bit integers using the Pytorch Quantization API and then partitioned into two sub-models: sub-model M_1, which contains layers 0 through $K - 1$, and sub-model M_2, which contains layers K through $K + L - 1$. The input video is divided into frames, which will serve as input to the network. Although network models typically handle frame extraction automatically during training and inference, in this work, the frames are manually extracted to provide more control over the data, as shown in Step 1 of Figure 3.1.

These frames are then used as input to the network model. Since the proposed compression method aims to explore the temporal correlation between the frames, and therefore the temporal correlation between the feature maps, two frames at different time steps—one at time $t - 1$ and the other at time t—are passed through the network, as shown in Step 2.

The $K - 1$ layer of the M_1 sub-model generates the feature maps, which are the outputs of the convolutional layers after the inference process, as shown in Step 3. These feature maps are then used as input for the encoding block in Figure 3.2.

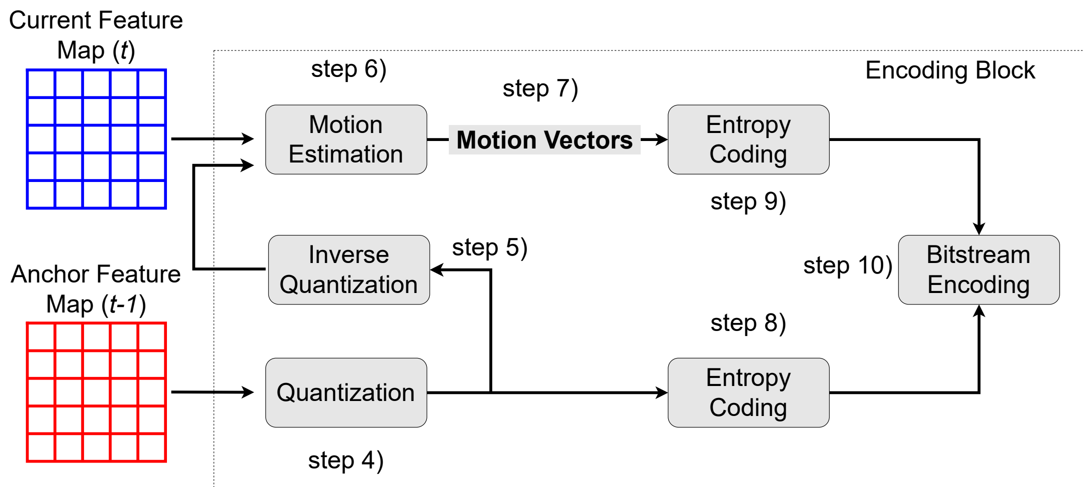

Figure 3.2: Encoding block of the proposed system.

The first step of the encoding block consists of the quantization of the feature map that will be used as the anchor, which is the feature map of the frame at time $t - 1$, as shown in Step 1 of Figure 3.2. This early quantization is essential for the algorithm, as it allows the prediction algorithm to perform motion estimation while accounting for the error introduced by the quantization on the encoder side, making the algorithm more robust.

The quantization scheme employed is an uniform quantization represented by the equa-

tion

$$Q(x, s, z) = \text{round}\left(\frac{x}{s} + z\right), \tag{3.1}$$

where s represents the scale or step of the quantizer, z is the quantized representation of zero, and x is the value being quantized. The quantization parameters s and z are determined using the same approach adopted by the PyTorch Quantization API, which uses observers—modules responsible for calculating the quantization parameters based on data statistics.

A simple approach is the *MinMax* observer, which ensures that the full range of the data is represented. The parameters are calculated as follows:

$$x_{\text{min}} = \min(X) \tag{3.2}$$

and

$$x_{\text{max}} = \max(X). \tag{3.3}$$

If the quantization scheme is symmetric—a method in which quantization levels are evenly distributed around zero, ensuring equal spacing for both positive and negative values—the parameters are calculated as follows:

$$s = \frac{2 \cdot \max\left(|x_{\text{min}}|, |x_{\text{max}}|\right)}{Q_{\text{max}} - Q_{\text{min}}} \tag{3.4}$$

and

$$z = 128, \tag{3.5}$$

where Q_{min} and Q_{max} represent the respective minimum and maximum integers that can be represented after quantization. If not otherwise specified, $Q_{\text{min}} = -127$ and $Q_{\text{max}} = 128$.

If the quantization scheme is affine—a method that applies a linear transformation (scaling and shifting) to map continuous values to discrete quantized levels—the parameters are calculated as follows:

$$s = \frac{x_{\text{max}} - x_{\text{min}}}{Q_{\text{max}} - Q_{\text{min}}} \tag{3.6}$$

and

$$z = Q_{\text{min}} - \text{round}\left(\frac{x_{\text{min}}}{s}\right). \tag{3.7}$$

Considering that the quantization parameters have already been calculated for 8-bit integers by the Pytorch Quantization API and that we want to find new parameters for the same data but with a smaller bit width, we assume the parameters are denoted as $s_{8\text{b}}$ and $z_{8\text{b}}$. Disregarding

rounding and considering the default Q_{min} and Q_{max} values, we have:

$$\frac{x}{s_{8\text{b}}} - z_{8\text{b}} = (\frac{x}{s_{\text{Nb}}} - z_{\text{Nb}}) \cdot 2^{N-8} \tag{3.8}$$

$$\frac{x}{s_{8\text{b}}} - z_{8\text{b}} = \frac{2^{N-8} \cdot x}{s_{\text{Nb}}} - 2^{N-8} \cdot z_{\text{Nb}}. \tag{3.9}$$

Through comparison:

$$s_{8\text{b}} = \frac{s_{\text{Nb}}}{2^{N-8}} \tag{3.10}$$

and

$$z_{8\text{b}} = z_{\text{Nb}} \cdot 2^{N-8}. \tag{3.11}$$

Then:

$$s_{\text{Nb}} = s_{8\text{b}} \cdot 2^{N-8} \tag{3.12}$$

and

$$z_{\text{Nb}} = z_{8\text{b}} \cdot 2^{8-N}. \tag{3.13}$$

If we wish to keep z as an integer, we have the following equations:

$$s_{\text{Nb}} = s_{8\text{b}} \cdot 2^{N-8} \tag{3.14}$$

and

$$z_{\text{Nb}} = \text{round}\left(z_{8\text{b}} \cdot 2^{8-N}\right). \tag{3.15}$$

Equations 3.14 and 3.15 are then used in the Step 4 of Figure 3.2 to determine the quantized values of the feature map, following Equation 3.1. In Step 5, the inverse quantization is performed using the following equation:

$$\hat{x} = (Q(x, s, z) - z) \cdot s. \tag{3.16}$$

The unquantized values of the anchor feature map, along with the values of the current feature map (which we aim to predict), are used as inputs to the motion estimation block, as illustrated in Step 6. The motion estimation employs the Three Step Search Algorithm, as described in Section 2.3.1.1, to generate the MVs in Step 7. The MVs and the quantized anchor feature map are then encoded using Huffman coding in Steps 8 and 9.

Unlike traditional prediction algorithms, this proposed method does not use the residual feature map (the difference between the current feature map and the predicted feature map based on the MVs). During algorithm evaluation, it was observed that the information in the residual feature map could be discarded without affecting the final predicted feature map at the decoder.

Thus, the proposed method reduces the amount of data transmitted by only sending the anchor feature map and the MVs, as shown in Step 10 of Figure 3.2.

The bit sequence is transmitted and used as input to the decoding block illustrated in Figure 3.3, where the bits are read and reconstructed, as shown in Step 11. The data is then decoded using Huffman coding in Step 12. Next, the quantized anchor feature map is processed through the inverse quantization block in Step 13. Finally, the decoded motion vectors and anchor feature map are used as inputs to the motion compensation block, as illustrated in Step 15.

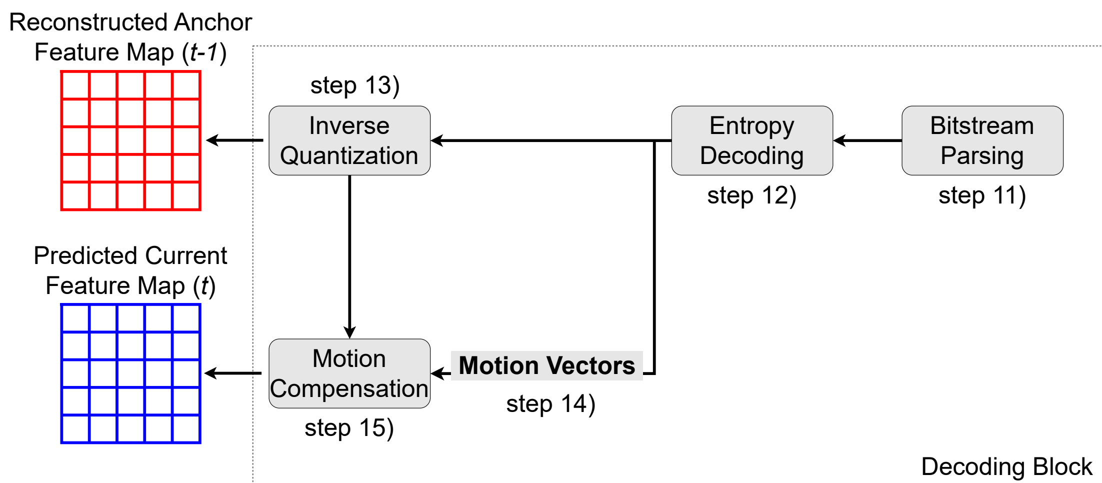

Figure 3.3: Decoding block of the proposed system.

In the motion compensation process, the TSS Algorithm uses the MVs to determine where each block in the current feature map should be retrieved from the anchor feature map. For each block, the MV indicates the displacement from its original position, guiding the decoder to copy the corresponding block from the anchor feature map. These blocks are then assembled according to their positions in the current feature map, forming the predicted feature map.

These feature maps are then used as input of the remaining layers of the network model as shown in Step 16 of Figure 3.1. The network than performs the inference and outputs the frames with the bouding boxes and the class of the objects identified in the frames, as shown in Step 17. The frames are then stacked to generate the final video output with the object detection bouding boxes, illustrated in Step 18 of Figure 3.1.

The proposed system described above predicts the current feature map at time t based on the anchor feature map at time $t-1$. This type of prediction is called unidirectional prediction (or one-way prediction), where only the past data is necessary to make a prediction. In video compression, a common approach is to use bi-directional prediction (or bidirectional predictive coding), where the information at time t is predicted based on the information at times $t-1$ and $t+1$.

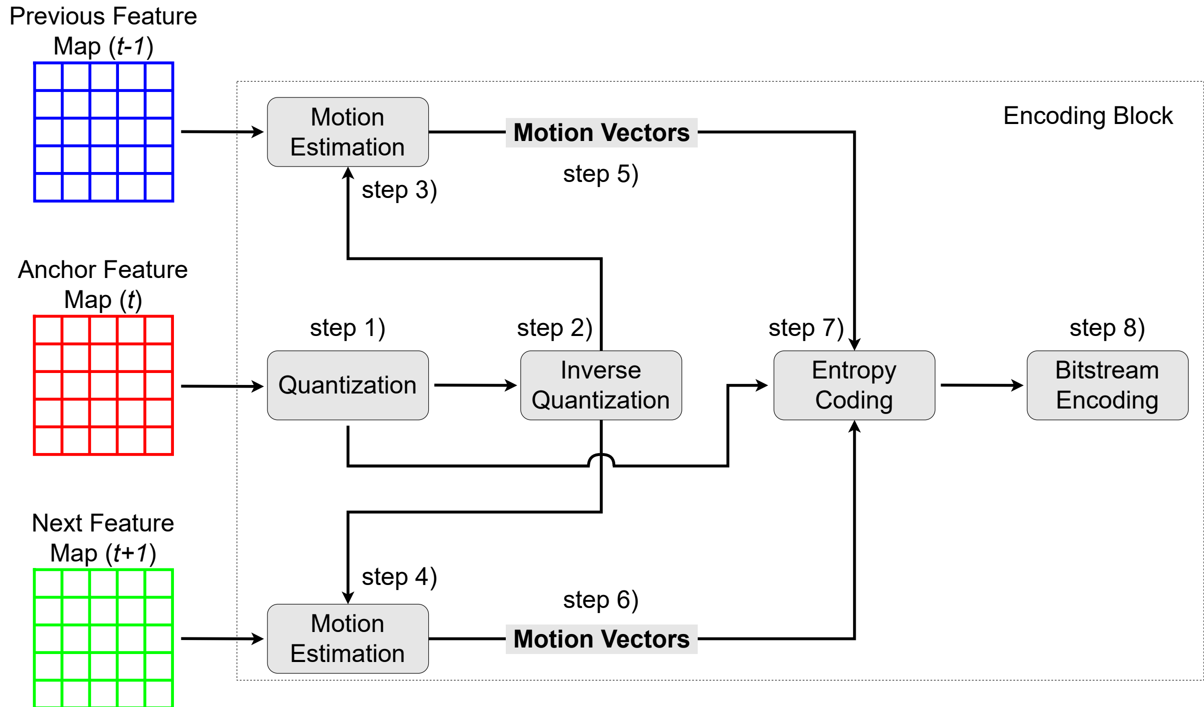

Figure 3.4: Encoding block of the proposed bi-directional prediction.

However, in the proposed system, a different form of bi-directional prediction is introduced. Instead of using two feature maps at different times to predict a single feature map, we use a single feature map to predict two other feature maps. Figure 3.4 illustrates the encoding block of the proposed bi-directional prediction. The feature map at time t serves as the anchor for the feature maps at times $t-1$ and $t+1$. The process is similar to that explained in Figure 3.2, where MVs are calculated for both feature maps, encoded using Huffman coding, and then sent to the decoder.

Figure 3.5 illustrates the decoding block of the proposed bi-directional prediction, where the process is similar to that described in Figure 3.3. The bi-directional prediction used in video compression enables more accurate reconstruction of frames by leveraging information from both temporal directions. However, the proposed bi-directional prediction allows for greater

data reduction during transmission or storage, as the feature map matrices can be represented as MVs, which are simpler to compress and lighter in storage space.

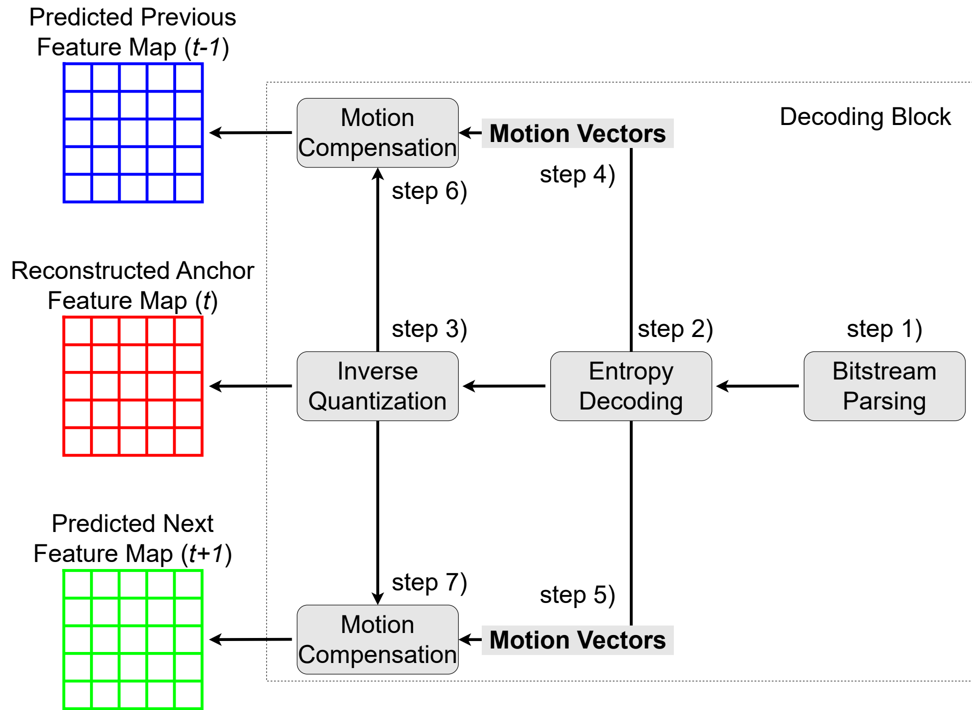

Figure 3.5: Decoding block of the proposed bi-directional prediction.

Chapter 4

Experimentals and Results

4.1 Network Models

4.1.1 Faster R-CNN with ResNet-50 Backbone

Faster R-CNN is a two-stage object detection model. In the first stage, it identifies regions in the image that are likely to contain objects using the Region Proposal Network (RPN). In the second stage, it classifies these regions and refines their bounding boxes. ResNet-50, a deep residual network with 50 layers, is widely used for feature extraction due to its strong representation capabilities [29] [30]. Figure 4.1 illustrates the architecture of Faster R-CNN with a ResNet-50 backbone.

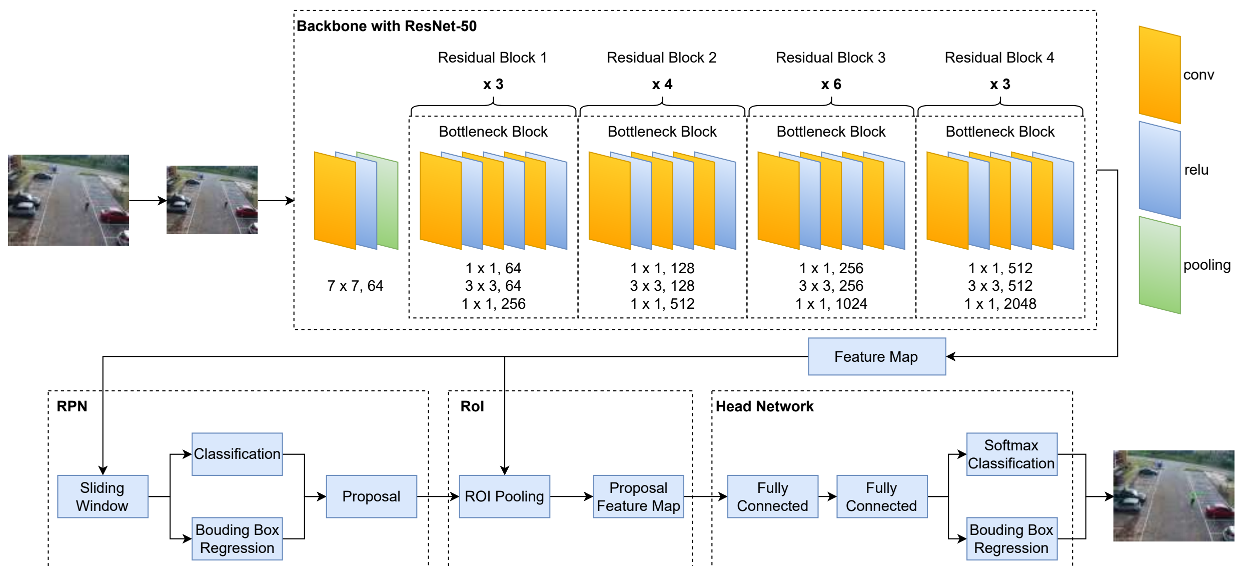

Figure 4.1: Architecture of the Faster R-CNN with ResNet-50 Backbone.

The structure of the Faster R-CNN with a ResNet-50 backbone can be divided into four main blocks:

- **1. Backbone (ResNet-50):**
  - Conv1: The initial 7×7 convolution with 64 filters, stride 2, and padding 3. Followed by batch normalization and ReLU activation. This reduces the spatial dimensions by half.
  - MaxPool: A 3×3 max pooling layer with stride 2, further reducing the spatial dimensions.
  - Residual Block 1: 3 residual bottleneck blocks, each containing a 1×1 convolution (to reduce channels), a $3x3$ convolution, and a final 1×1 convolution (to restore channels). Each block has a skip connection to help with gradient flow.
  - Residual Block 2: 4 residual bottleneck blocks. The first block down-samples the feature map, and the subsequent blocks maintain the same dimensions.
  - Residual Block 3: 6 residual bottleneck blocks, with the first block down-sampling.
  - Residual Block 4: 3 residual bottleneck blocks, with the first block down-sampling. The final output of Residual Block 4 is a high-level feature map (usually $7 \times 7 \times 2048$ for a 224×224 input).
  - Output Feature Map: The output from Residual Block 4 (typically $7 \times 7 \times 2048$) is fed to the Region Proposal Network.
- **2. Region Proposal Network (RPN):**
  - 3×3 Conv Layer: A convolutional layer with 512 filters to process the feature map from the backbone.
  - Anchor Generation: At each spatial location, the RPN generates anchor boxes with different sizes and aspect ratios (e.g., 3 scales and 3 aspect ratios, resulting in 9 anchors per location).
  - Objectness Score Head: A 1×1 convolution that outputs a binary score for each anchor (object or background).
  - Bounding Box Regression Head: A 1×1 convolution that outputs bounding box adjustments (dx, dy, dw, dh) for each anchor.

  - Output: The RPN outputs a set of proposals (likely regions containing objects) with coordinates and objectness scores.
- **3. RoI Pooling:**
  - Projects each proposal onto the feature map.
  - Divides each proposal into a fixed grid (e.g., 7×7) and applies max pooling within each bin.
  - Produces a fixed-size feature map ($7 \times 7 \times C$) for each proposal, which is passed to the Head Network.
- **4. Head Network:**
  - Two Fully Connected Layers: Each RoI feature map is passed through two fully connected layers to generate high-level features for classification and bounding box regression.
  - Classification Head: A softmax layer that outputs the probability distribution over classes (including the background).
  - Bounding Box Regression Head: A regression layer that refines the bounding box coordinates for each proposal.

ResNet-50 offers several advantages, including high accuracy due to its deep structure, which enables it to learn complex patterns and improve object detection accuracy. The use of residual connections helps avoid the vanishing gradient problem, allowing the network to be deeper and more accurate. Additionally, it provides good generalization for large objects, effectively handling complex backgrounds and large-scale objects through its multi-scale feature representation.

Since the proposed method focuses on compressing the feature maps extracted, the partition points of the network are located within the backbone. Each convolution and activation pair generates a feature map that can be compressed using the proposed algorithm, resulting in 16 possible partition points within the backbone. However, for this analysis, only four partition points were selected, as they effectively represent the different alternatives:

- **layer-1-block-2:** This partition point is located at the end of Residual Block 1, where the feature maps are the output of the last bottleneck block, with a depth of 256.

- **layer-2-block-3:** This partition point is located at the end of Residual Block 1, where the feature maps are the output of the last bottleneck block, with a depth of 512.
- **layer-3-block-5:** This partition point is located at the end of Residual Block 1, where the feature maps are the output of the last bottleneck block, with a depth of 1024.
- **layer-4-block-2:** This partition point is located at the end of Residual Block 1, where the feature maps are the output of the last bottleneck block, with a depth of 2048.

4.1.2 Faster R-CNN with MobileNetV3 Backbone

Faster R-CNN with MobileNetV3 is a lightweight version of Faster R-CNN, optimized for efficiency and speed. MobileNetV3 incorporates techniques such as depthwise separable convolutions and inverted residual blocks, which significantly reduce computation while maintaining reasonable accuracy. This version of Faster R-CNN is well-suited for real-time applications or for use in mobile and embedded systems [31]. Figure 4.2 illustrates the architecture of Faster R-CNN with a MobileNetV3 backbone.

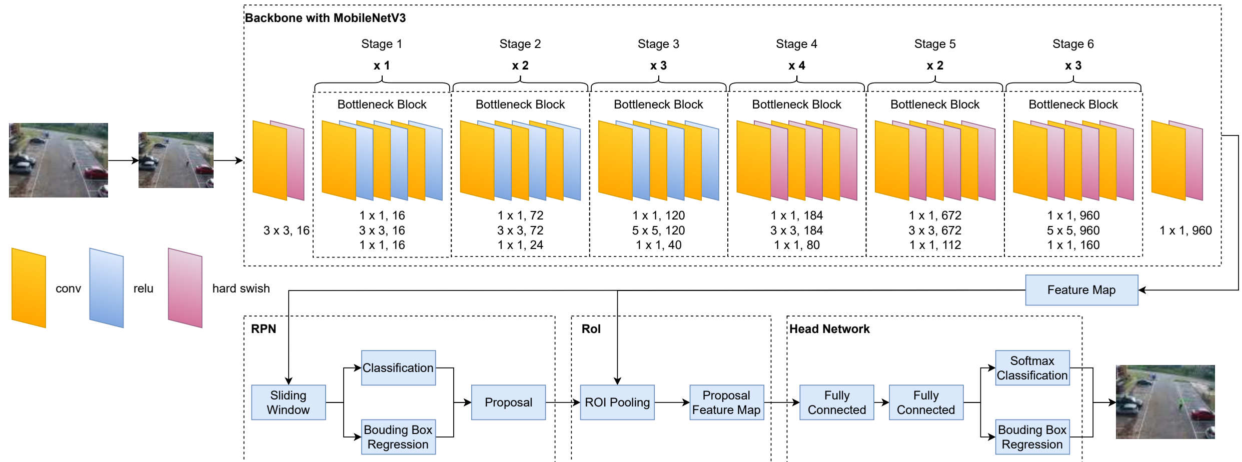

Figure 4.2: Architecture of the Faster R-CNN with MobileNetV3 Backbone.

The structure of the Faster R-CNN with a MobileNetV3 backbone can be divided into four main blocks:

- **1. Backbone (MobileNetV3 Large):**
  - Initial Convolution: A 3×3 convolution with 16 filters, stride 2, and padding 1, followed by batch normalization and a hard-swish activation.

  - Each bottleneck block consists of an expansion layer (to increase channels), a depthwise convolution (to capture spatial features independently for each channel), and a projection layer (to reduce the channels).
  - Stage 1: First bottleneck block with 16 output channels and no expansion.
  - Stage 2: Two bottleneck blocks with 24 output channels and an expansion factor of 4.
  - Stage 3: Three bottleneck blocks with 40 output channels and an expansion factor of 3.
  - Stage 4: Four bottleneck blocks with 80 output channels and an expansion factor of 6.
  - Stage 5: Two bottleneck blocks with 112 output channels and an expansion factor of 6.
  - Stage 6: Three bottleneck blocks with 160 output channels and an expansion factor of 6.
  - Conv Head: A final 1×1 convolution with 960 filters to produce the final feature map.
  - Output Feature Map: The final feature map output from MobileNetV3 Large is typically $7 \times 7 \times 960$.

- **2. Region Proposal Network (RPN):**
  - 3×3 Conv Layer: A convolutional layer with 512 filters to process the feature map from the backbone.
  - Anchor Generation: At each spatial location, the RPN generates anchor boxes with different sizes and aspect ratios (e.g., 3 scales and 3 aspect ratios, resulting in 9 anchors per location).
  - Objectness Score Head: A 1×1 convolution that outputs a binary score for each anchor (object or background).
  - Bounding Box Regression Head: A 1×1 convolution that outputs bounding box adjustments (dx, dy, dw, dh) for each anchor.
  - Output: The RPN outputs a set of proposals (likely regions containing objects) with coordinates and objectness scores.

- **3. RoI Pooling:**
  - Projects each proposal onto the feature map.
  - Divides each proposal into a fixed grid (e.g., 7×7) and applies max pooling within each bin.
  - Produces a fixed-size feature map ($7 \times 7 \times C$) for each proposal, which is passed to the Head Network.
- **4. Head Network:**
  - Two Fully Connected Layers: Each RoI feature map is passed through two fully connected layers to generate high-level features for classification and bounding box regression.
  - Classification Head: A softmax layer that outputs the probability distribution over classes (including the background).
  - Bounding Box Regression Head: A regression layer that refines the bounding box coordinates for each proposal.

MobileNetV3's lightweight architecture enhances the efficiency of this version of Faster R-CNN, making it faster and more computationally efficient. Its lower latency enables real-time capability, making it suitable for applications in mobile or embedded systems. Additionally, the depthwise convolutions in MobileNetV3 effectively capture details of smaller objects, especially in lower-resolution inputs, providing good performance for detecting small objects.

Similar to the network model with a ResNet-50 backbone, each convolution and activation pair in the MobileNetV3 backbone generates a feature map that can be compressed using the proposed algorithm, resulting in 16 possible partition points within the backbone. However, for this analysis, only seven partition points were selected:

- **layer-1-block-1:** This partition point is located at the end of Stage 1, where the feature maps are the output of the last bottleneck block, with a depth of 16.
- **layer-3-block-2:** This partition point is located at the end of Stage 2, where the feature maps are the output of the last bottleneck block, with a depth of 24.
- **layer-6-block-3:** This partition point is located at the end of Stage 3, where the feature maps are the output of the last bottleneck block, with a depth of 40.

- **layer-10-block-2:** This partition point is located at the end of Stage 4, where the feature maps are the output of the last bottleneck block, with a depth of 80.

- **layer-12-block-3:** This partition point is located at the end of Stage 5, where the feature maps are the output of the last bottleneck block, with a depth of 112.

- **layer-15-block-3:** This partition point is located at the end of Stage 6, where the feature maps are the output of the last bottleneck block, with a depth of 160.

- **layer-16-block-2:** This partition point is located at the end of the backbone, where the feature maps are the output of the last bottleneck block, with a depth of 960.

4.1.3 RetinaNet with ResNet-50 Backbone

RetinaNet is a single-stage object detection model. Unlike Faster R-CNN, it does not rely on a region proposal network; instead, it predicts object classes and bounding boxes in a single pass over the image. RetinaNet introduces Focal Loss, which emphasizes hard-to-classify examples during training, making it effective for handling class imbalance in detection tasks [32]. Figure 4.3 illustrates the architecture of RetinaNet with a ResNet-50 backbone.

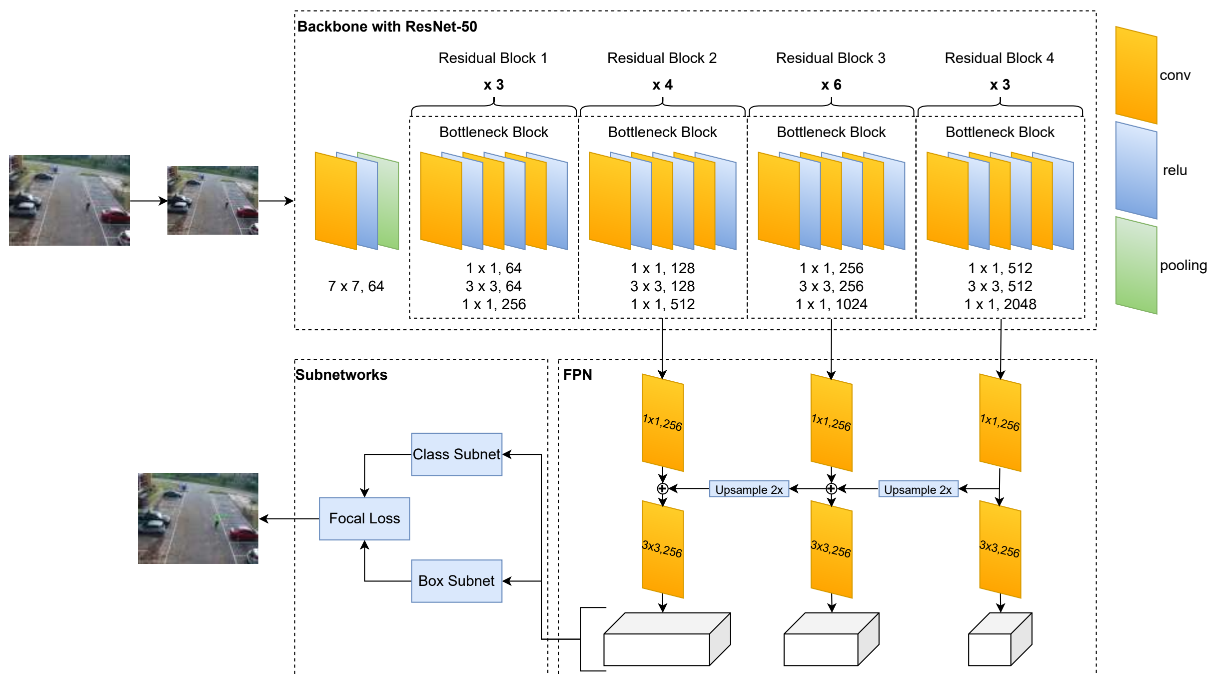

Figure 4.3: Architecture of the RetinaNet with ResNet-50 Backbone.

The structure of the RetinaNet with a ResNet-50 backbone can be divided into three main blocks:

- **1. Backbone (ResNet-50):**
  - Conv1: The initial 7×7 convolution with 64 filters, stride 2, and padding 3. Followed by batch normalization and ReLU activation. This reduces the spatial dimensions by half.
  - MaxPool: A 3×3 max pooling layer with stride 2, further reducing the spatial dimensions.
  - Residual Block 1: 3 residual bottleneck blocks, each containing a 1×1 convolution (to reduce channels), a $3x3$ convolution, and a final 1×1 convolution (to restore channels). Each block has a skip connection to help with gradient flow.
  - Residual Block 2: 4 residual bottleneck blocks. The first block down-samples the feature map, and the subsequent blocks maintain the same dimensions.
  - Residual Block 3: 6 residual bottleneck blocks, with the first block down-sampling.
  - Residual Block 4: 3 residual bottleneck blocks, with the first block down-sampling. The final output of Residual Block 4 is a high-level feature map (usually $7 \times 7 \times 2048$ for a 224×224 input).
  - Output Feature Map: The output from Residual Block 4 (typically $7 \times 7 \times 2048$) is fed to the Region Proposal Network.
- **2. Feature Pyramid Network (FPN):**
  - The FPN takes feature maps from multiple levels in the backbone (Residual Blocks 2, 3 and 4 of ResNet-50) and combines them into a multi-scale feature pyramid.
  - Each feature map from the backbone is passed through 1×1 and 3×3 convolutions to create a pyramid of feature maps at different scales.
  - This multi-scale pyramid allows RetinaNet to detect objects of varying sizes by providing multiple spatial resolutions for feature extraction.
  - Output Dimensions: The FPN outputs feature maps at various resolutions (e.g., 56×56, 28×28, 14×14, etc.), all with the same depth (e.g., 256 channels).

- **3. Subnets for Classification and Box Regression:**
  - Classification Subnet: The classification subnet applies a series of $3x3$ convolutions to each level of the feature pyramid to predict object classes at each spatial location. A final sigmoid layer generates binary class scores for each anchor, which are processed by Focal Loss. The output dimension consists of class scores for each anchor box across all feature map levels.
  - Box Regression Subnet: The box regression subnet applies a series of 3x3 convolutions to each level of the feature pyramid to predict bounding box coordinates for each anchor. The output dimension consists of bounding box coordinates (dx, dy, dw, dh) for each anchor across all feature map levels.

RetinaNet offers several advantages, including the use of Focal Loss, which focuses on hard examples, addresses class imbalance, and improves accuracy for rare objects. Its single-stage detection architecture enables faster inference by removing the need for a separate RPN, making it suitable for real-time applications. Additionally, the efficient multi-scale detection provided by the FPN enables robust detection of objects at various scales, making RetinaNet effective for detecting objects of varying sizes.

Since this model also uses ResNet-50 as the backbone, the partition points are the same as those described for the Faster R-CNN with a ResNet-50 backbone:

- **layer-1-block-2:** This partition point is located at the end of Residual Block 1, where the feature maps are the output of the last bottleneck block, with a depth of 256.
- **layer-2-block-3:** This partition point is located at the end of Residual Block 1, where the feature maps are the output of the last bottleneck block, with a depth of 512.
- **layer-3-block-5:** This partition point is located at the end of Residual Block 1, where the feature maps are the output of the last bottleneck block, with a depth of 1024.
- **layer-4-block-2:** This partition point is located at the end of Residual Block 1, where the feature maps are the output of the last bottleneck block, with a depth of 2048.

4.1.4 Quantization of Models

Quantization in PyTorch is a model optimization technique designed to reduce the computational and memory demands of neural networks by representing weights and activations

with lower precision, often 8-bit integers instead of the usual 32-bit floating points. This approach makes models more efficient for inference, particularly on edge devices and in real-time applications where resources are limited [33].

PyTorch's quantization API supports three main methods of quantization: Dynamic Quantization, Static (Post-Training) Quantization, and Quantization-Aware Training (QAT). Each method offers a different balance of accuracy, performance, and ease of implementation, allowing developers to choose the one that best suits their model's deployment needs.

4.1.4.1 Dynamic Quantization

Dynamic quantization is the simplest form of quantization in PyTorch and is particularly well-suited for models with fully connected layers, LSTMs, and GRUs, where activation values can vary significantly at runtime.

In this approach, weights are pre-quantized to 8-bit integers, which drastically reduces model size and memory usage, while activations remain in their original 32-bit floating-point format until runtime, where they are dynamically quantized on a per-batch basis based on their current range. This dynamic quantization allows the model to adapt to changes in activation ranges due to different inputs, making it more flexible than other quantization methods.

While dynamic quantization does not achieve the same level of latency reduction as other methods, it is ideal for use cases that are memory-bound rather than compute-bound, such as NLP tasks where models often need large embedding matrices. Since it doesn't need calibration data, it is also a convenient option for quick optimizations without additional data collection or fine-tuning.

4.1.4.2 Static (Post-Training) Quantization

Static quantization, also known as post-training quantization, is a more robust quantization method that targets both weights and activations, quantizing them to 8-bit integers. This approach significantly reduces both the memory footprint and computational cost, as lower-precision operations consume less power and allow for higher processing speeds.

To achieve accurate static quantization, a calibration step is required: a small but representative dataset is passed through the model to gather statistics on the ranges of activations. These ranges are then used to set the quantization parameters (scales and zero points) for each layer, ensuring that the quantization captures the most informative parts of the data distribution.

Static quantization can achieve substantial speedups and memory savings, especially on convolutional networks, which are computation-heavy. This method is ideal for deployment on devices like mobile phones or IoT devices where computational efficiency is paramount. While calibration data is required for accurate activation range estimation, the model does not need to be retrained, making static quantization efficient to implement once calibration data is available.

4.1.4.3 Quantization-Aware Training (QAT)

Quantization-Aware Training (QAT) is the most accurate form of quantization offered by PyTorch. In QAT, the model is trained with simulated quantization effects, allowing it to adapt to quantization noise during training.

This technique adds fake quantization operations to the forward pass, which apply quantization effects (like clamping and rounding) to weights and activations, while the backward pass uses the full-precision weights to optimize gradients. This balance enables the model to learn a robust representation that can tolerate quantization errors, reducing the loss of precision that often occurs with lower-bit representations.

QAT is computationally intensive, as it requires training with quantization effects applied, and typically demands a larger dataset and extended training time. However, it achieves the highest accuracy among quantized models, particularly valuable in mission-critical applications where accuracy cannot be sacrificed, such as medical imaging, autonomous driving, and financial models. QAT is particularly beneficial for complex architectures like large CNNs or transformers, where even slight improvements in quantization robustness can significantly affect real-world performance.

4.1.4.4 Implementation

The PyTorch Quantization API offers two different approaches for building and executing models: FX Graph Mode and Eager Mode. FX (Function Transformations) Graph Mode is a newer approach designed to facilitate advanced optimizations, transformations, and analysis on PyTorch models. It converts a model's operations into an intermediate representation (IR) graph, making it easier to understand, modify, and optimize the model's structure. Eager Mode, on the other hand, is the default execution mode in PyTorch, where operations are executed immediately as they are called. This mode emphasizes ease of use, dynamic behavior, and debugging simplicity, allowing developers to interact with the model in a step-by-step, imperative

fashion.

The existing quantized models and documentation available in the PyTorch API focus mainly on simple network models for image classification, necessitating the custom implementation of quantized versions for the models described above. While using FX Graph Mode is simpler—since PyTorch provides functions to perform quantization and dequantization on weights and activations—it became evident during implementation that partitioning the model at specific points was not possible with this approach.

Consequently, the models were quantized using Eager Mode, which introduced additional complexity due to the lack of documentation support for more complex models. Each model required redesigning to ensure quantization was applied to all layers within both the backbone and the rest of the network. It was necessary to fuse all layers within each residual block to align with the quantization functions. Additionally, some operations within the bottleneck and the RPN needed to be replaced. The implementation of the quantized models is available on GitLab.

4.2 Dataset and Correlation Analysis

The dataset used in this study was the UFPark dataset, as proposed in [34]. It comprises video clips from a surveillance camera capturing footage throughout the day. The dataset includes samples taken during the morning, afternoon, and night, totaling 32.77 GB of data and approximately 3.25 hours of footage. The video clips vary in length from 5 to 30 seconds, depending on activity patterns in the parking lot. In this work, we used sample video clips with a duration of 10 seconds at a rate of 30 frames per second, resulting in a total of 300 frames per clip.

For the correlation analysis of the frames and the feature maps, we compared the images and observed metrics like the Structural Similarity Index (SSIM) and MSE, along with visualizations such as a correlation heatmap and residual analysis.

SSIM is a perceptual metric that compares two images based on how humans perceive visual structure. SSIM evaluates similarity by analyzing luminance, contrast, and structural patterns in both images, giving a score between -1 and 1. A higher SSIM value (closer to 1) indicates higher similarity and visual quality, making it particularly useful in applications like compression and restoration, where preserving image details is important.

MSE metric quantifies the average squared difference between pixel values of two images. This metric reflects the absolute differences in pixel intensity, where a lower MSE value indicates higher similarity. Although simple and effective, MSE does not account for perceptual qualities, so even small differences in structure or texture may produce large MSE values, so it is often used alongside perceptual metrics like SSIM for a more balanced analysis.

The residual analysis visually displays the absolute pixel-by-pixel difference between images, highlighting areas where they diverge most. This residual image uses intensity differences to emphasize structural dissimilarities, aiding in identifying regions of change or variation.

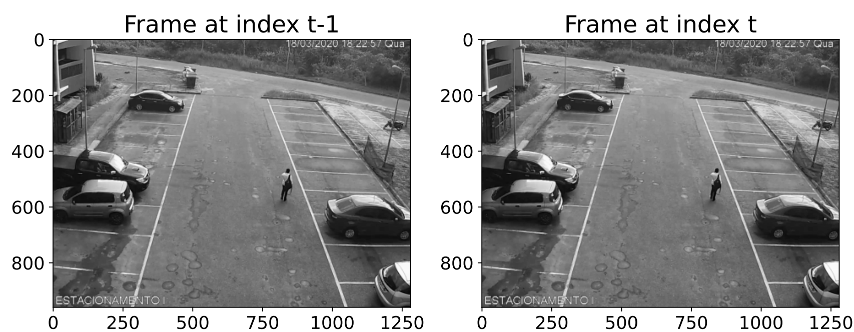

Figure 4.4: Comparison between frames at index $t - 1$ and t.

Figure 4.4 illustrates the comparison between the frame at index $t - 1$ and the frame at index t. The obtained values of SSIM and MSE are 0.994 and 1.621, respectively, indicating a high temporal correlation between the frames, as expected. Figure 4.5 illustrates the residual between the frames, where the movement of the person is more visible.

Figure 4.5: Residual between frames.

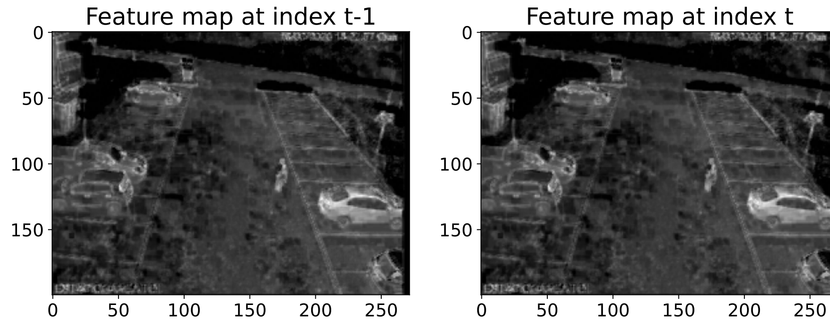

Figure 4.6: Comparison between feature maps at first partition point.

The core of the proposed method is the existence of temporal correlation between the feature maps, which assumes that the temporal correlation observed between the frames extends to the feature maps within the layers of the neural network. Figure 4.6 illustrates the comparison between the feature maps extracted from the first partition point of the Faster R-CNN with a ResNet-50 backbone (as described in Section 4.1.1), where the output dimensions are $200 \times 272 \times 256$. The obtained values of SSIM and MSE are 0.995 and 1.622, respectively, indicating a high temporal correlation between the feature maps. Figure 4.7 illustrates the residual information.

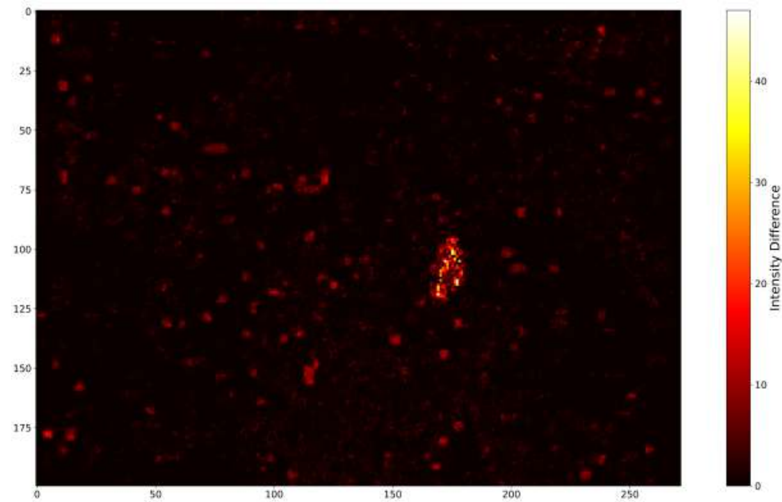

Figure 4.7: Residual between feature maps at first partition point.

For the second partition point, Figure 4.8 illustrates the comparison between the feature maps. At this point, the feature maps have dimensions of $100 \times 136 \times 512$, with SSIM and MSE values of 0.996 and 0.764, respectively. This also indicates a high temporal correlation between the feature maps. Figure 4.9 illustrates the residual information.

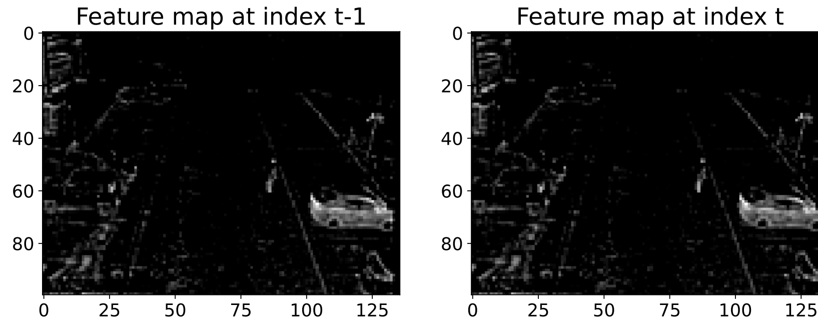

Figure 4.8: Comparison between feature maps at second partition point.

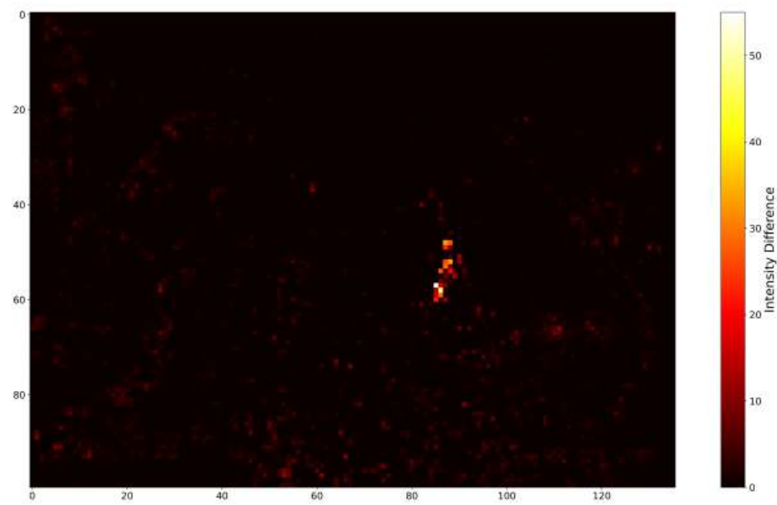

Figure 4.9: Residual between feature maps at second partition point.

For the third partition point, Figure 4.10 shows the comparison between the feature maps (with dimensions of $50 \times 68 \times 1024$), where the obtained SSIM and MSE values are 0.991 and 2.601. Despite the increase in MSE, these values still represent a high temporal correlation. Figure 4.11 illustrates the residual information.

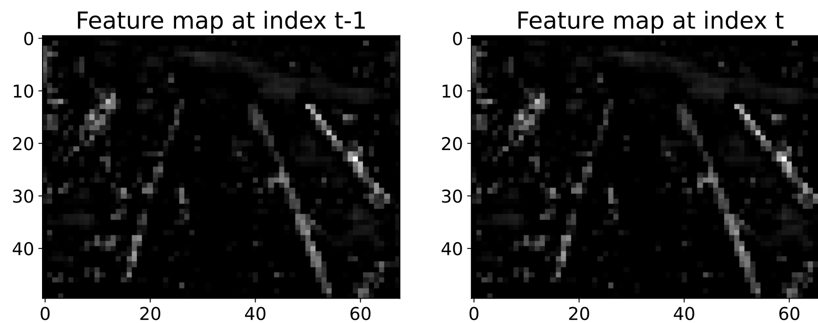

Figure 4.10: Comparison between feature maps at third partition point.

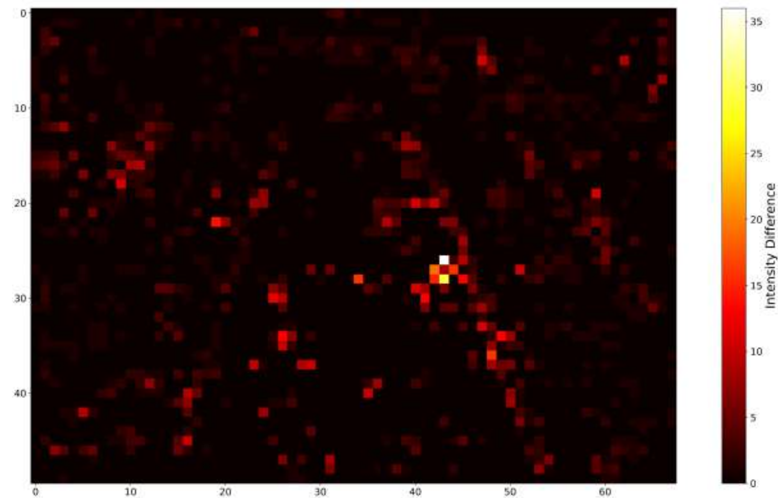

Figure 4.11: Residual between feature maps at third partition point.

Finally, for the last partition point, Figure 4.12 illustrates the comparison between the feature maps ($25 \times 34 \times 2048$), with SSIM and MSE values of 0.985 and 3.481, respectively, also indicating a high temporal correlation. Figure 4.13 illustrates the residual information. As we can see, the residual at this partition point contains less information, making the proposed algorithm more efficient, since we are not sending the residual.

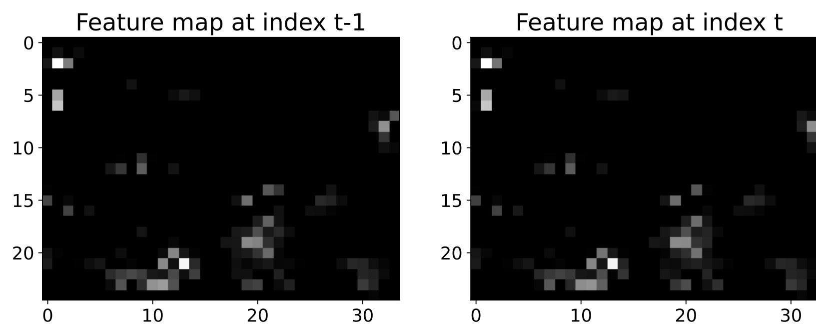

Figure 4.12: Comparison between feature maps at fourth partition point.

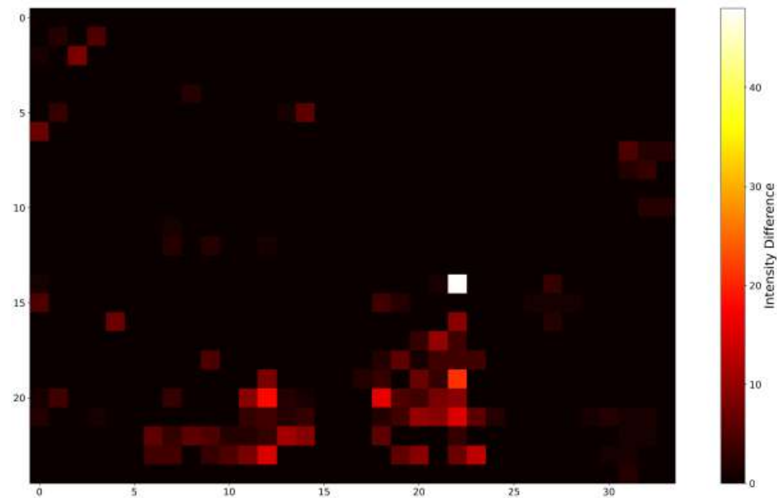

Figure 4.13: Residual between feature maps at fourth partition point.

4.3 Results

This section describes the experiments performed and the results obtained after applying the proposed system to the different network architectures and the dataset described.

The network models were pre-trained on the COCO dataset [35] and quantized using the Static (Post-Training) Quantization method (as described in Section 4.1.4.2), which allows for quantizing both weights and activations without the need to retrain the model. For the experiments, three samples of 10 seconds each were used, extracted from different times of the day, as detailed in Section 4.2. Table 4.1 presents the possible combinations of experiments for each video sample.

	Sample Input		
Network Model	**Sample 1**	**Sample 2**	**Sample 3**
Faster R-CNN with ResNet-50	8	8	8
Faster R-CNN with MobileNetV3	14	14	14
RetinaNet with ResNet-50	8	8	8

Table 4.1: Total number of experiments for each video sample.

Since we are analyzing how the prediction of feature maps reduces the amount of data to be transmitted and how it affects the network's final inference, the values in the table represent the total number of experiments required to gather this information. To calculate this,

we multiply the number of possible partition points by 2, representing the metrics for rate and mAP.

mAP [36] [37] [38] is a key metric for evaluating the performance of object detection models, especially when the model must predict both the location and class of objects within an image. mAP is derived from Average Precision (AP), which measures the model's accuracy across different levels of recall, capturing its balance between precision and recall. Precision is defined as the proportion of correctly identified objects out of all predictions, while recall is the proportion of actual objects correctly identified. These metrics are calculated as follows:

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}} \tag{4.1}$$

and

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}} \tag{4.2}$$

To compute AP, precision is evaluated at various recall levels, typically by plotting a precision-recall curve and calculating the area under this curve. Mathematically, AP can be expressed as the integral of precision $P(r)$ over recall *r* from 0 to 1:

$$\text{AP} = \int_0^1 P(r)\, dr. \tag{4.3}$$

In practice, AP is often approximated by averaging precision values at specific recall thresholds. Once AP values are calculated for each class, mAP is obtained by averaging these AP values across all *N* object classes:

$$\text{mAP} = \frac{1}{N} \sum_{i=1}^{N} \text{AP}_i. \tag{4.4}$$

The mAP metric can be reported at different Intersection over Union (IoU) thresholds, commonly *mAP@0.50* and *mAP@0.75*. IoU measures the overlap between the predicted bounding box and the ground truth bounding box:

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}. \tag{4.5}$$

Evaluating mAP at multiple IoU thresholds provides a fuller picture of the model's detection quality, capturing both its ability to find objects (through *mAP@0.50*) and localize them accurately (through *mAP@0.75*). This way, mAP ensures that the model is not only finding objects but also placing bounding boxes precisely around them, which is crucial in applications like autonomous driving, medical imaging, and surveillance.

Each experiment shown in Table 4.1 results in a total of 250 GB of data to be stored and analyzed, consisting of the feature maps at four different partition points and for each bit depth (ranging from 4 to 8). Therefore, this section will focus on the results obtained from Sample 1 using Faster R-CNN with a ResNet-50 backbone, as these results are sufficient to support the main conclusions and comparisons. Table 4.2 shows all parameters that can be modified and the values set for this experiment.

Partition Point	Block Size	Number of bits	Scale at 8 bits	Zero points at 8 bits
layer-1-block-2	8x8	4 to 8	0.024894	41
layer-2-block-3	8x8	4 to 8	0.082080	30
layer-3-block-5	4x4	4 to 8	0.041199	35
layer-4-block-2	4x4	4 to 8	0.092679	38

Table 4.2: Parameter values for Sample 1 using Faster R-CNN with a ResNet-50 backbone.

Figure 4.14 illustrates the data rates at the first partition point. At the top of the graph, we see the rate required to transmit the extracted feature maps without any compression scheme, which leads to a rate of approximately 418 MBps, assuming 30 frames per second. Applying uniform quantization to the feature maps (using parameters from the PyTorch API) results in a 2x compression ratio at 4 bits, with a rate of 208 MBps, as shown by curve 2 in the graph. The same rate is achieved when using uniform quantization with ZVC, as shown in curve 3. For uniform quantization with Huffman coding, we achieve a compression ratio of approximately 6x at 4 bits, with a rate of 73 MBps, as seen in curve 4.

The proposed unidirectional prediction method is illustrated in curves 5 and 6, where, at 4 bits in the quantizer, we reach a compression ratio of 4x, with a rate of 105 MBps for both unidirectional TSS alone and unidirectional TSS with ZVC coding. The proposed bi-directional prediction method is shown in curves 7 and 8, where, at 4 bits in the quantizer, we achieve a 6x compression ratio, with a rate of 70 MBps for both bi-directional TSS alone and with ZVC coding. Curves 9 and 10 represent the proposed unidirectional TSS and bi-directional TSS with Huffman coding, where, at 4 bits, we reach compression ratios of approximately 11x and 16x, with rates of 37 MBps and 25 MBps, respectively.

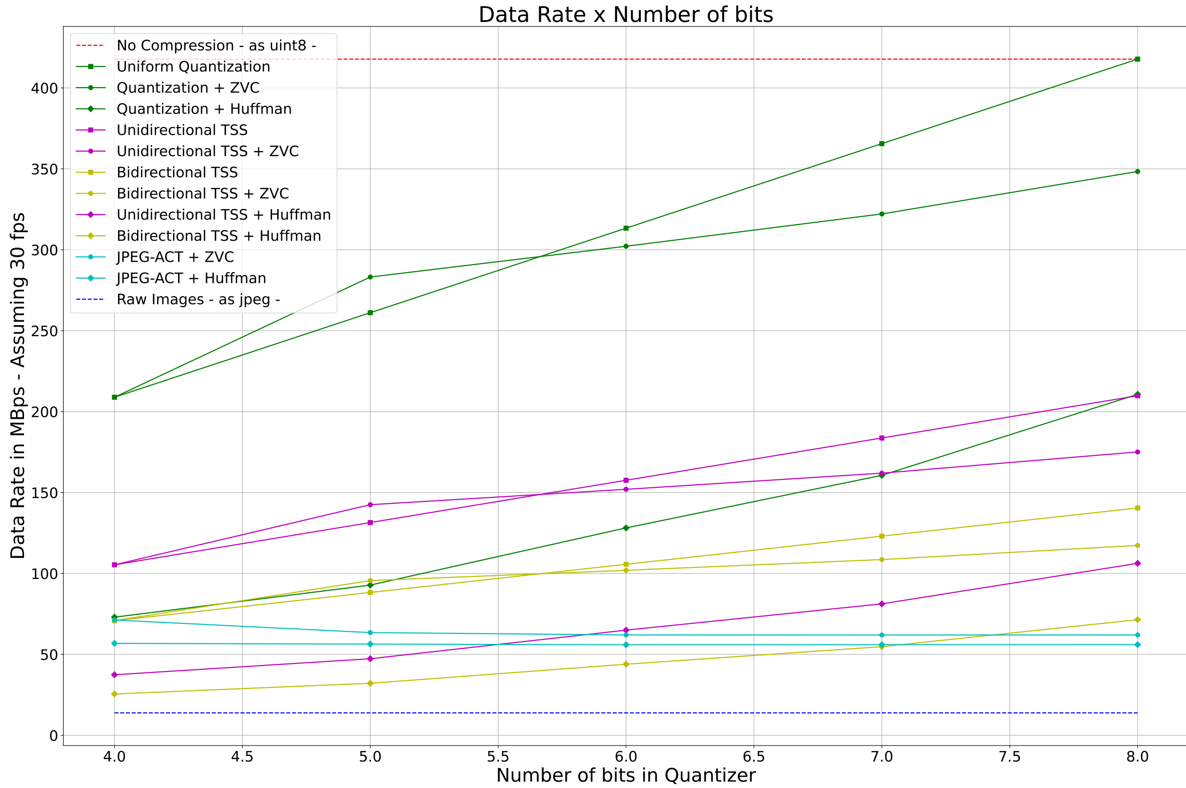

Figure 4.14: Data rate at the first partition point.

Curves 11 and 12 represent the method proposed in [39] for compressing feature maps, achieving, at 4 bits in the quantizer, compression ratios of around 5x and 7x, with rates of 71 MBps and 56 MBps, respectively. Finally, the last curve shows the rate for transmitting raw frames as JPEG, which is 13 MBps.

As shown, for the first partition point, the proposed method achieves a significant rate reduction compared to the feature maps without any compression, demonstrating its efficiency in handling the information contained within the feature maps. However, when compared to the rate required for frame transmission, the proposed method does not achieve lower rates. This is due to the high volume of information carried by the feature maps in the initial network layers, which have dimensions of $200 \times 272 \times 256$ representing a total of $13.926.400$ values, compared to the $3.686.400$ values ($1280 \times 960 \times 3$) in a frame.

Figure 4.15 illustrates the mAP of the network when running inference with the proposed prediction method. The dotted red line represents the mAP of the network without any compression, reaching a value of 96%. After applying Static (Post-Training) Quantization to quantize the network to 8-bit integers, the mAP drops to 94%, as shown by the dotted blue line.

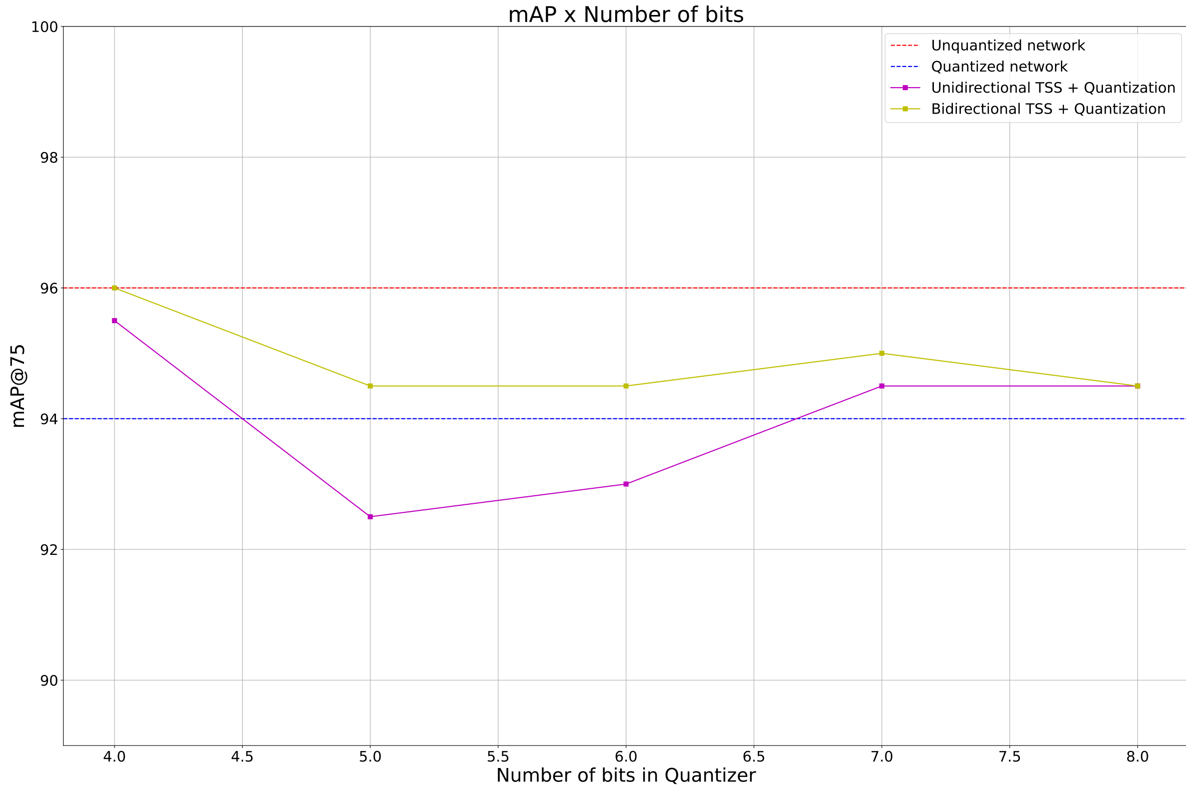

Figure 4.15: mAP at the first partition point.

After applying the proposed method to the feature maps, with the quantizer bit depth varying from 4 to 8 bits, the yellow and purple lines show the resulting mAP. It is important to note that the network was quantized only to 8-bit integers, as supported by the PyTorch API. Therefore, the feature maps were re-quantized to 8-bit integers by the PyTorch API before being fed back into the network.

As shown by the yellow and purple lines, the obtained mAP varies by 1% to 2% compared to the mAP of the quantized network. These variations are due to a random process in the proposed method, which can add or remove information from the feature maps, potentially increasing or decreasing the final model accuracy.

Figure 4.16 illustrates the data rates at the second partition point. The data rates follow a similar pattern to those at the first partition point, with the rate required to transmit the extracted feature maps without any compression at the top of the graph, approximately 208 MBps. Next, we see uniform quantization alone and with ZVC coding, achieving compression ratios of 2x and 4x, with rates of 104 MBps and 46 MBps, respectively.

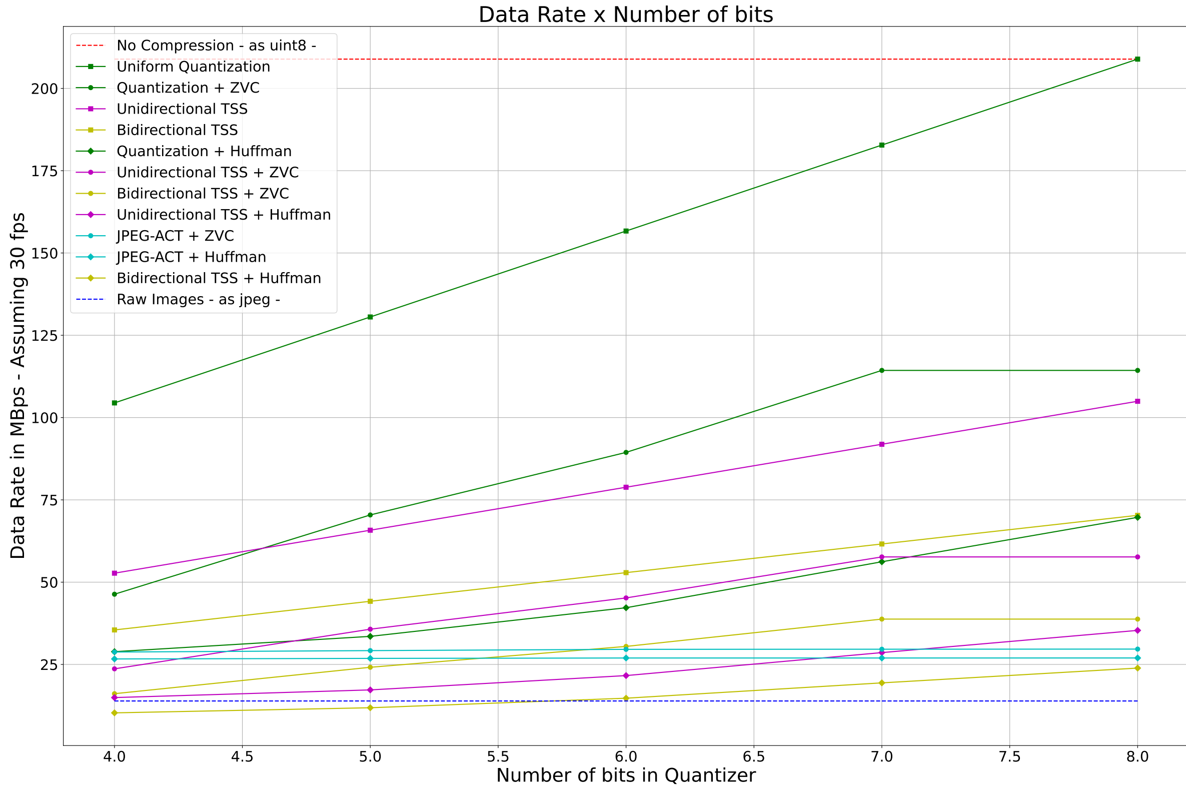

Figure 4.16: Data rate at the second partition point.

Curves 4 and 5 represent the proposed unidirectional and bi-directional methods without additional coding, achieving compression ratios of 4x and 5x at 4 bits, with rates of approximately 52 MBps and 35 MBps, respectively. Curves 6, 10, and 11 show similar compression ratios of approximately 8x, with rates around 28 MBps, 28 MBps, and 26 MBps, respectively.

The remaining curves represent the proposed prediction methods. Notably, bi-directional prediction with Huffman coding achieves a compression ratio of 20x compared to uncompressed feature maps, reducing the transmission rate from 13 MBps (for JPEG-compressed raw frames) to 10 MBps. As we progress deeper into the network layers, the feature maps carry less information, making the proposed method more efficient.

Figure 4.17 illustrates the mAP, where we observe a similar random variation. For quantizer bit depths of 5, 6, 7, and 8, the mAP deviates by less than 1% compared to the quantized network's mAP. However, at a 4-bit quantizer, there is a notable accuracy drop of around 3%. Depending on the application, this accuracy loss may lead to detection issues, although it did not significantly impact the final results in the object detection experiments.

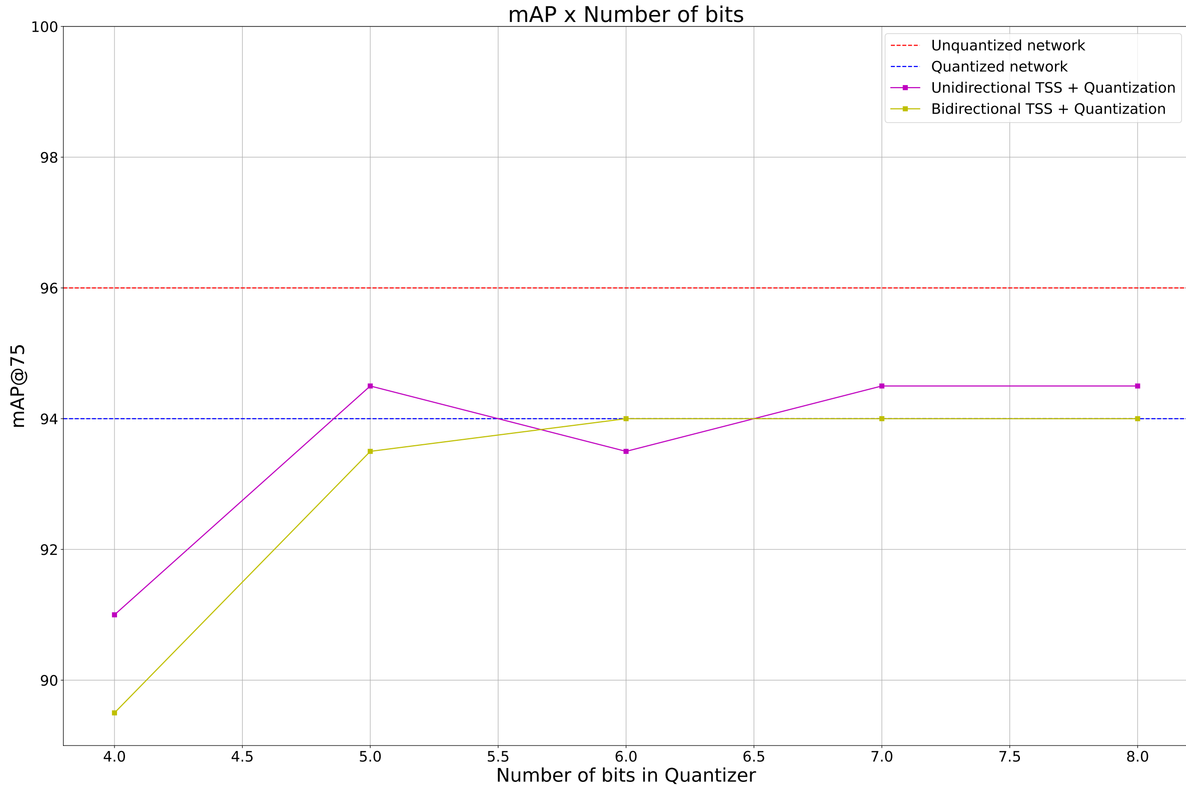

Figure 4.17: mAP at the second partition point.

Figure 4.18 illustrates the data rates at the third partition point. The data rates follow a similar pattern to those at the previous partition points. However, at this partition point, instead of only one curve below the rate required for raw frames, we observe five curves. Here, the proposed unidirectional and bi-directional methods with ZVC and Huffman coding achieve rates lower than that required to transmit the raw frames as JPEG.

For unidirectional prediction with ZVC and Huffman coding, we observe compression ratios of approximately 8x and 12x, respectively, compared to uncompressed feature maps. This results in a rate reduction from 13 MBps to 12 MBps and 8.5 MBps, respectively, compared to raw frames. Similarly, for bi-directional prediction with ZVC and Huffman coding, the compression ratios are approximately 11x and 15x, respectively, compared to uncompressed feature maps, and reducing the rate from 13 MBps to 9 MBps and 6.6 MBps, respectively, compared to raw frames.

Figure 4.19 illustrates the mAP, where we also observe a similar random variation, with around 2% variance in final accuracy. This variation did not significantly impact the results in the object detection experiments.

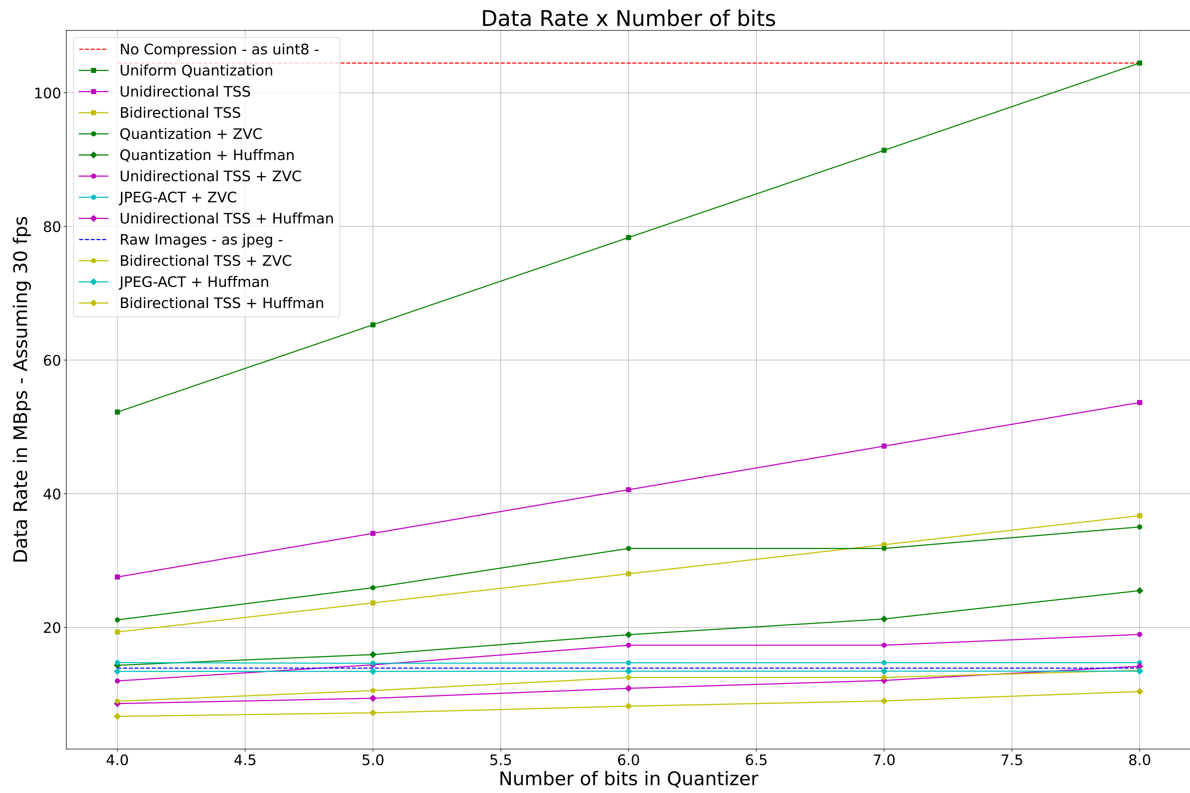

Figure 4.18: Data rate at the third partition point.

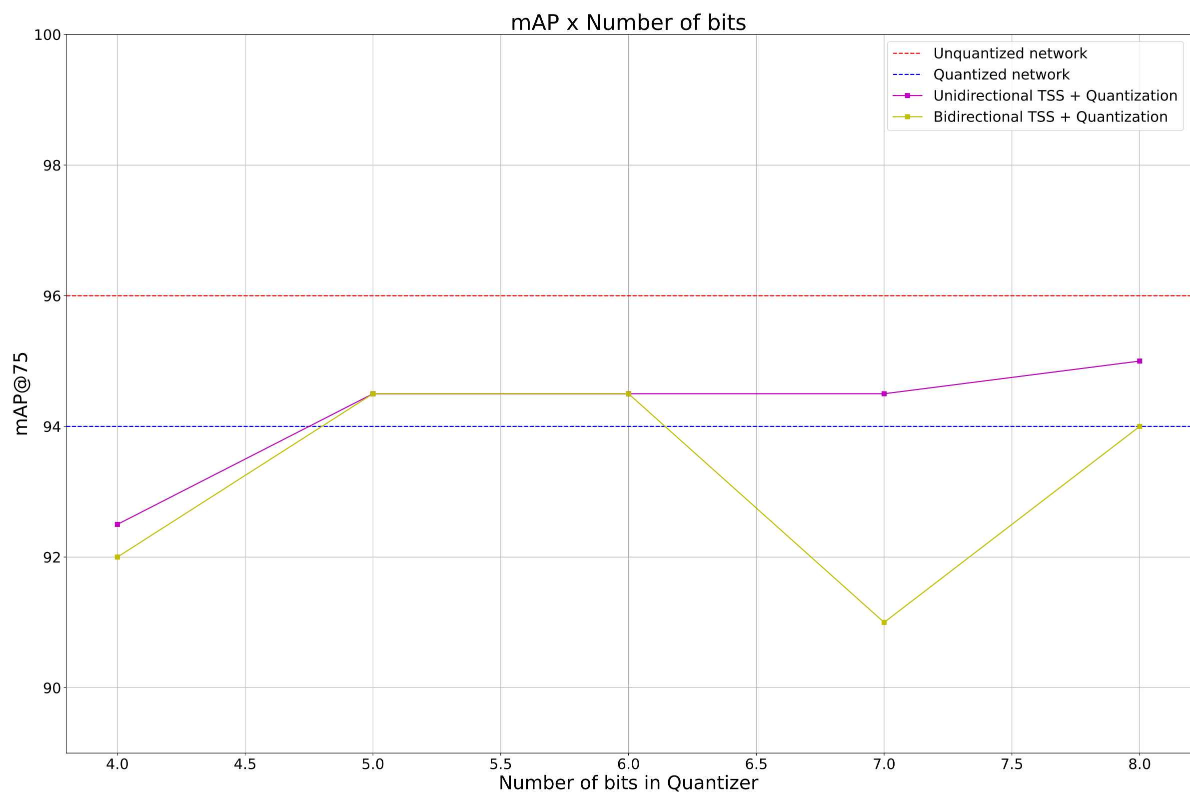

Figure 4.19: mAP at the third partition point.

Finally, Figure 4.20 illustrates the data rates at the last partition point. At the top of the graph, we see the rate required to transmit the extracted feature maps without any compression, which is approximately 52 MBps. Applying uniform quantization to the feature maps results in a 2x compression ratio at 4 bits, with a rate of 26 MBps, as shown by curve 2 in the graph. Curve 3 represents unidirectional prediction, achieving a compression ratio of 4x with a rate of 13 MBps.

Curves 4 and 5 represent uniform quantization with ZVC coding and bi-directional prediction, with compression ratios of approximately 6x and 5x, respectively, and rates of 8 MBps and 9 MBps.

For unidirectional prediction with ZVC and Huffman coding, we observe compression ratios of approximately 10x and 12x, respectively, compared to uncompressed feature maps, with rates of 5 MBps and 4.2 MBps. Compared to the raw frames, these achieve compression ratios of approximately 2.5x and 3x. Similarly, for bi-directional prediction with ZVC and Huffman coding, the compression ratios are approximately 13x and 15x, reducing the rate from 52 MBps to 4 MBps and 3.4 MBps, respectively, compared to uncompressed feature maps. Relative to the raw frames, these methods achieve compression ratios of approximately 3.25x and 3.8x, significantly reducing the amount of information to be transmitted.

Figure 4.21 illustrates the mAP, where we also observe similar random variation, with around 1% variance in final accuracy, which did not significantly impact the results in the object detection experiments.

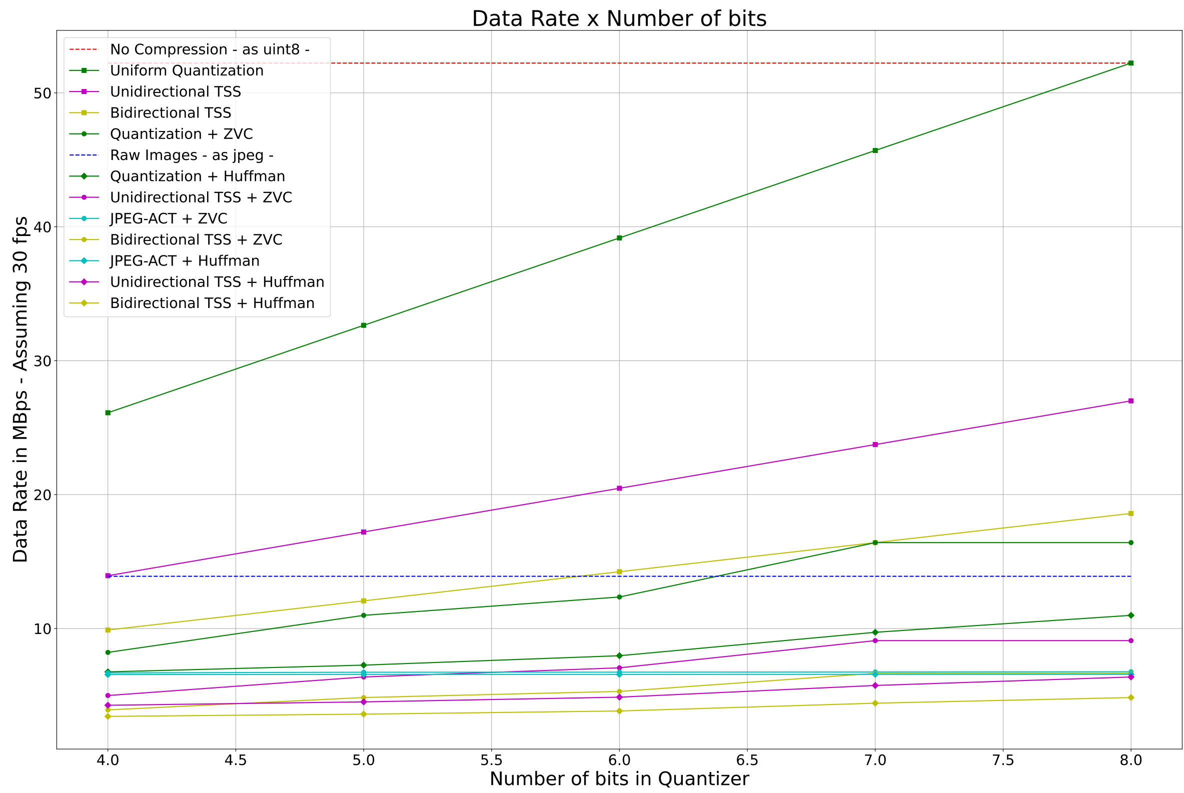

Figure 4.20: Data rate at the last partition point.

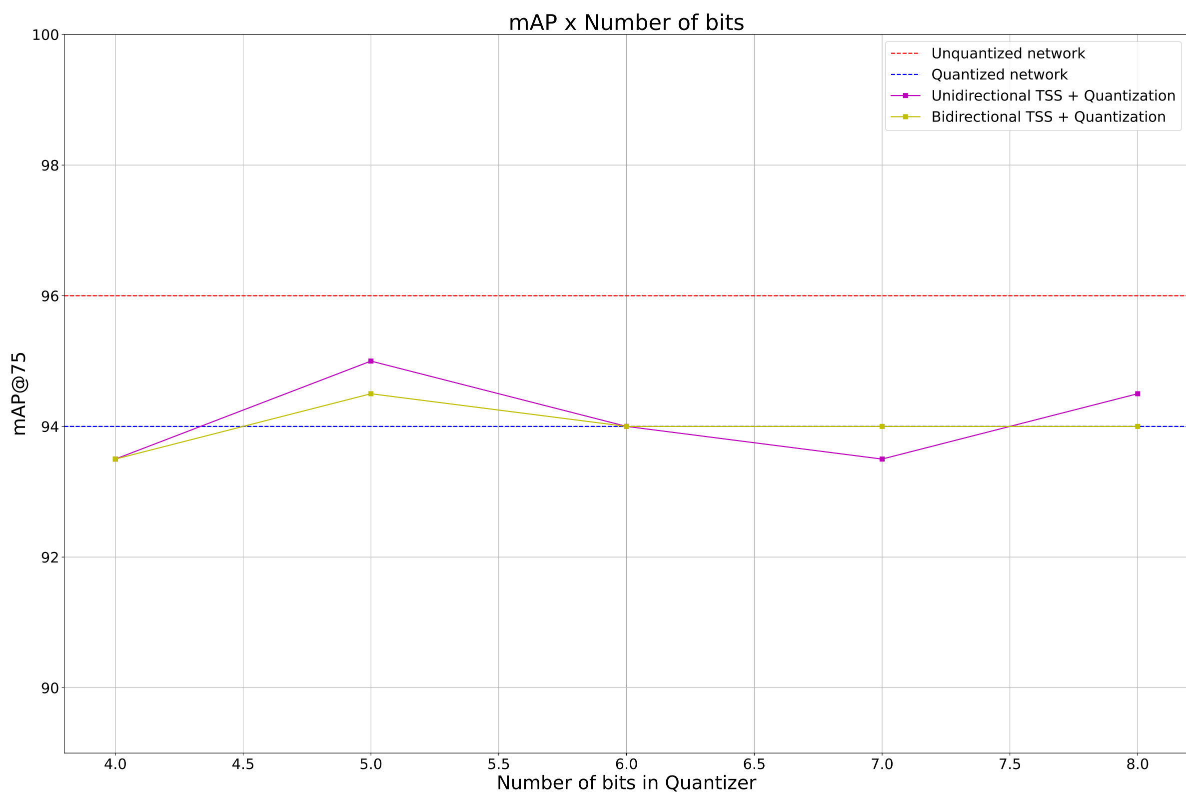

Figure 4.21: mAP at the last partition point.

Chapter 5

Conclusion and Future Work

This dissertation proposed a system consisting of two main elements: the first is the implementation of a partitioning strategy for a neural network used in video-based object detection, targeting applications that require such networks on devices with limited computational capacity. To address these constraints, the network architecture is organized such that the initial layers are allocated on the end device, while the remaining layers are processed on a cloud server. The second element is a compression method that explores temporal correlation between feature maps, significantly reducing the amount of information that needs to be stored and transmitted between the end device and the server. This approach respects hardware limitations and the transmission rate constraints of the communication channel.

In the initial chapters, a literature review was conducted on the main components of a convolutional neural network, covering the input layer, convolutional and pooling layers, which are responsible for feature extraction and dimensionality reduction from the input data, and concluding with the fully connected layers, which are responsible for the final weight training of the network for the inference process. Following this, some of the primary activation functions used in conjunction with the network layers were introduced, aimed at increasing the model's expressive capacity and enabling the network to embody artificial intelligence. Additionally, the main techniques for data compression were presented, exploring both spatial and temporal redundancy in the information to be compressed.

The results obtained with the UFPark dataset and the Faster R-CNN with a ResNet-50 backbone model demonstrated that the proposed system exhibited minimal accuracy loss compared to the accuracy of the quantized model across the four defined partitioning points. At the first partitioning point, using 4 bits to represent the data, the system achieved a mAP of ap-

proximately 95%, with a compression ratio of about 16x relative to uncompressed data. At the second point, we observed the lowest mAP of 89.5%, with a compression ratio of 20x compared to uncompressed data. At the third partitioning point, the system achieved a mAP of 92%, with a compression ratio of approximately 15x compared to uncompressed data and about 2x compared to sending the raw frames as JPEG. Finally, at the last partitioning point, a mAP of 93.5% was achieved, with a compression ratio of approximately 15x compared to uncompressed data and 3.8x compared to raw frames. Thus, the proposed model successfully meets the expected results, minimally affecting the model's accuracy while significantly reducing the amount of data that needs to be stored and transmitted between the end device and the cloud server, and adding a layer of security to the collected data.

For future work, the correlation between feature maps can be further explored, specifically the correlation between kernels, to concentrate the most relevant information in the initial depth layers of the feature maps, thereby further reducing the amount of data to be stored. For instance, in the initial layers where feature maps have dimensions of $200 \times 272 \times 256$, it may be possible to reduce the last dimension and focus relevant information in the initial layers. Additionally, extending this research to align more closely with federated learning [40] concepts is of interest, allowing the proposed compression system to be validated in scenarios with multiple clients.

Bibliography

[1] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy *et al.*, "Evolving Deep Neural Networks," in *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Elsevier, 2024, pp. 269–287.

[2] W. Chen, X. Lin, J. Lee, A. Toskala, S. Sun, C. F. Chiasserini, and L. Liu, "5G-Advanced Toward 6G: Past, Present, and Future," *IEEE Journal on Selected Areas in Communications*, vol. 41, no. 6, pp. 1592–1619, 2023.

[3] B. Mishra, D. Garg, P. Narang, and V. Mishra, "Drone-Surveillance for Search and Rescue in Natural Disaster," *Computer Communications*, vol. 156, pp. 1–10, 2020.

[4] Y. Matsubara, D. Callegaro, S. Singh, M. Levorato, and F. Restuccia, "Bottlefit: Learning Compressed Representations in Deep Neural Networks for Effective and Efficient Split Computing," in *2022 IEEE 23rd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. IEEE, 2022, pp. 337–346.

[5] Z. Shao, X. Chen, L. Du, L. Chen, Y. Du, W. Zhuang, H. Wei, C. Xie, and Z. Wang, "Memory-Efficient CNN Accelerator Based on Interlayer Feature Map Compression," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 69, no. 2, pp. 668–681, 2021.

[6] C. Xie, Z. Shao, N. Zhao, Y. Du, and L. Du, "An Efficient CNN Inference Accelerator Based on Intra- and Inter-Channel Feature Map Compression," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2023.

[7] H. Zhang, L. Liu, H. Zhou, L. Si, H. Sun, and N. Zheng, "FCHP: Exploring the Discriminative Feature and Feature Correlation of Feature Maps for Hierarchical DNN Pruning and

Compression," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 32, no. 10, pp. 6807–6820, 2022.

[8] Z. Zhang, M. Wang, M. Ma, J. Li, and X. Fan, "MSFC: Deep Feature Compression in Multi-Task Network," in *2021 IEEE International Conference on Multimedia and Expo (ICME)*. IEEE, 2021, pp. 1–6.

[9] K. J. Cios, "Deep Neural Networks—A Brief History," in *Advances in Data Analysis with Computational Intelligence Methods*. Springer, 2018, pp. 183–200.

[10] W. Samek, G. Montavon, S. Lapuschkin, C. J. Anders, and K.-R. Müller, "Explaining Deep Neural Networks and Beyond: A Review of Methods and Applications," *Proceedings of the IEEE*, vol. 109, no. 3, pp. 247–278, 2021.

[11] A. A. M. Al-Saffar, H. Tao, and M. A. Talab, "Review of Deep Convolution Neural Network in Image Classification," in *2017 International Conference on Radar, Antenna, Microwave, Electronics, and Telecommunications (ICRAMET)*. IEEE, 2017, pp. 26–31.

[12] J. Wu, "Introduction to Convolutional Neural Networks," *National Key Lab for Novel Software Technology. Nanjing University. China*, vol. 5, no. 23, p. 495, 2017.

[13] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects," *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[14] H. S. Das and P. Roy, "A Deep Dive Into Deep Learning Techniques for Solving Spoken Language Identification Problems," in *Intelligent Speech Signal Processing*. Elsevier, 2019, pp. 81–100.

[15] Y. Wang, Y. Li, Y. Song, and X. Rong, "The Influence of the Activation Function in a Convolution Neural Network Model of Facial Expression Recognition," *Applied Sciences*, vol. 10, no. 5, p. 1897, 2020.

[16] K. Sayood, *Introduction to Data Compression*. Morgan Kaufmann, 2017.

[17] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression*. Springer Science & Business Media, 2012, vol. 159.

[18] H. K. Joy, M. R. Kounte, A. Chandrasekhar, and M. Paul, "Deep Learning Based Video Compression Techniques with Future Research Issues," *Wireless Personal Communications*, vol. 131, no. 4, pp. 2599–2625, 2023.

[19] S. Pandit, P. K. Shukla, A. Tiwari, P. K. Shukla, M. Maheshwari, and R. Dubey, "Review of Video Compression Techniques Based on Fractal Transform Function and Swarm Intelligence," *International Journal of Modern Physics B*, vol. 34, no. 08, p. 2050061, 2020.

[20] B. Furht, J. Greenberg, and R. Westwater, *Motion Estimation Algorithms for Video Compression*. Springer Science & Business Media, 2012, vol. 379.

[21] A. Barjatya, "Block Matching Algorithms for Motion Estimation," *IEEE Transactions Evolution Computation*, vol. 8, no. 3, pp. 225–239, 2004.

[22] S. Kulkarni, D. Bormane, and S. Nalbalwar, "Coding of Video Sequences Using Three Step Search Algorithm," *Procedia Computer Science*, vol. 49, pp. 42–49, 2015.

[23] R. Srinivasan and K. Rao, "Predictive Coding Based on Efficient Motion Estimation," *IEEE Transactions on Communications*, vol. 33, no. 8, pp. 888–896, 1985.

[24] Y.-W. Huang, C.-Y. Chen, C.-H. Tsai, C.-F. Shen, and L.-G. Chen, "Survey on Block Matching Motion Estimation Algorithms and Architectures with New Results," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 42, pp. 297–320, 2006.

[25] C.-F. Tseng, Y.-T. Lai, and M.-J. Lee, "A VLSI Architecture for Three-Step Search with Variable Block Size Motion Vector," in *The 1st IEEE Global Conference on Consumer Electronics 2012*. IEEE, 2012, pp. 628–631.

[26] M. Sharma *et al.*, "Compression Using Huffman Coding," *IJCSNS International Journal of Computer Science and Network Security*, vol. 10, no. 5, pp. 133–141, 2010.

[27] M. K. Mathur, S. Loonker, and D. Saxena, "Lossless Huffman Coding Technique for Image Compression and Reconstruction Using Binary Trees," *International Journal of Computer Technology and Applications*, vol. 3, no. 1, 2012.

[28] S. Akhter and M. Haque, "ECG Compression Using Run Length Encoding," in *2010 18th European Signal Processing Conference*. IEEE, 2010, pp. 1645–1649.

[29] B. W. Choi, S. Kang, H. W. Kim, O. D. Kwon, H. D. Vu, and S. W. Youn, "Faster Region-Based Convolutional Neural Network in the Classification of Different Parkinsonism Patterns of the Striatum on Maximum Intensity Projection Images of [18F] FP-CIT Positron Emission Tomography," *Diagnostics*, vol. 11, no. 9, p. 1557, 2021.

[30] Q.-Q. Zhou, J. Wang, W. Tang, Z.-C. Hu, Z.-Y. Xia, X.-S. Li, R. Zhang, X. Yin, B. Zhang, and H. Zhang, "Automatic Detection and Classification of Rib Fractures on Thoracic CT Using Convolutional Neural Network: Accuracy and Feasibility," *Korean Journal of Radiology*, vol. 21, no. 7, p. 869, 2020.

[31] M. Abd Elaziz, A. Dahou, N. A. Alsaleh, A. H. Elsheikh, A. I. Saba, and M. Ahmadein, "Boosting COVID-19 Image Classification Using MobileNetV3 and Aquila Optimizer Algorithm," *Entropy*, vol. 23, no. 11, p. 1383, 2021.

[32] H. Tian, Y. Zheng, and Z. Jin, "Improved RetinaNet Model for the Application of Small Target Detection in the Aerial Images," in *IOP Conference Series: Earth and Environmental Science*, vol. 585, no. 1. IOP Publishing, 2020, p. 012142.

[33] Pytorch, "Pytorch Quantization: https://pytorch.org/docs/stable/quantization.html."

[34] I. Nascimento, P. Castro, S. Klautau, L. Gonçalves, F. Brito, A. Klautau, S. Lins *et al.*, "Public Dataset of Parking Lot Videos for Computational Vision Applied to Surveillance," in *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2020, pp. 61–64.

[35] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common Objects in Context," in *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*. Springer, 2014, pp. 740–755.

[36] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes (VOC) Challenge," *International Journal of Computer Vision*, vol. 88, pp. 303–338, 2010.

[37] M. Everingham, S. A. Eslami, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes Challenge: A Retrospective," *International Journal of Computer Vision*, vol. 111, pp. 98–136, 2015.

[38] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, pp. 211–252, 2015.

[39] R. D. Evans, L. Liu, and T. M. Aamodt, "JPEG-ACT: Accelerating Deep Learning Via Transform-Based Lossy Compression," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 860–873.

[40] J. Wen, Z. Zhang, Y. Lan, Z. Cui, J. Cai, and W. Zhang, "A Survey on Federated Learning: Challenges and Applications," *International Journal of Machine Learning and Cybernetics*, vol. 14, no. 2, pp. 513–535, 2023.