A MASTER THESIS ON

Simultaneous Multispectral Detection of Objects

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

Diplom-Ingenieur
(Equivalent to Master of Science)

in

Embedded Systems (066 504)

by

Thomas Kotrba

01525909

Supervisor(s):
Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch
Dipl.-Ing. Martin Lechner

Vienna, Austria
June 2023

Abstract

Object detection in multispectral images, e.g., visible and infrared light, can benefit real-world applications such as autonomous driving or traffic surveillance. This is due to complementary information, especially in adverse weather and low illumination conditions. In the case of deep-learning-based object detectors, this complementary information can be fused at several positions throughout the network architecture. The main differences between the approaches are how good the performance is compared to the single spectrum reference networks and how high the latency increase is due to the additional network operations. Although there are many works on feature fusion in neural networks, there is rarely a focus on embedded hardware performance. This thesis compares the impact of different fusion architectures in a deep-learning-based object detector and optimizes these fusion architectures for an embedded device. Particularly, six different fusion architectures are proposed and optimized for an NVIDIA Jetson AGX Xavier, and their inference time, power consumption, and object detection performance are compared. The results show that the proposed multispectral fusion approaches outperform the reference networks in object detection metrics compared to the baseline networks. The early fusion approaches only lead to a reasonably slight increase in latency.

Kurzfassung

Die Objekterkennung mit multispektralen Bildern, z.B. im sichtbaren und infraroten Spektrum, kann bei realen Anwendungen wie dem autonomen Fahren oder der automatischen Verkehrsüberwachung vorteilhaft sein. Der Grund dafür ist, dass die Informationen der multispektralen Spektren einander ergänzen, insbesondere beim Einfluss von unvorteilhaftem Wetter oder schlechter Ausleuchtung. Im Falle von Deep-Learning-basierten Objektdetektoren können diese komplementären Informationen an verschiedenen Stellen in der Netzwerkarchitektur zusammengeführt werden. Die Hauptunterschiede zwischen den Ansätzen äußern sich, wie gut die Objektdetektion im Vergleich zu den (monospektralen) Referenznetzwerken ist und wie hoch der Anstieg der Latenzzeit aufgrund der zusätzlichen Netzwerkoperationen ist. Es gibt zahlreiche Arbeiten zur Feature-Fusionierung in neuronalen Netzen, aber kaum welche, die sich mit den Auswirkungen auf Embedded Hardware beschäftigen. Diese Arbeit vergleicht die Auswirkungen verschiedener Fusionsarchitekturen in einem Deep-Learning-basierten Objektdetektor und optimiert diese Fusionsarchitekturen für ein Embedded Device. Es werden sechs verschiedene Fusionsarchitekturen vorgestellt und für einen NVIDIA Jetson AGX Xavier optimiert. Anschließend werden sie bezüglich ihrer Netzwerklatenz, dem Energieverbrauch und ihre Objekterkennungsqualität verglichen. Die Ergebnisse zeigen, dass die vorgeschlagenen multispektralen Fusionsansätze die Referenznetzwerke in Bezug auf die Objekterkennung der ursprünglichen Netzwerke übertreffen, während die Fusionsansätze zu Beginn des Netzwerks nur zu einem relativ geringen Anstieg der Latenzzeit führen.

Preface

All of the work presented henceforth was conducted in the Christian Doppler Laboratory for Embedded Machine Learning at the TU Wien, Vienna, Austria. I did the writing and carried out all experiments.

An extended abstract about this work's main results was accepted as

> Thomas Kotrba, Martin Lechner, Omair Sarwar and Axel Jantsch - ”Multispectral Feature Fusion for Deep Object Detection on Embedded NVIDIA Platforms“ 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE), Antwerp, Belgium, 2023.

It will be published in the conference proceedings. I was responsible for writing the manuscript and presenting the work on the 19th of April, 2023 at the conference in Antwerp, Belgium. Martin Lechner, Omair Sarwar and Axel Jantsch were involved throughout the work in the manuscript formation, revising the work critically, and approving the version to be published.

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Copyright Statement

I, Thomas Kotrba, hereby declare that this thesis is my own original work and, to the best of my knowledge and belief, it does not:

- Breach copyright or other intellectual property rights of a third party.
- Contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
- Contain material which to a substantial extent has been accepted for the qualification of any other degree or diploma of a university or other institution of higher learning.
- Contain substantial portions of third party copyright material, including but not limited to charts, diagrams, graphs, photographs or maps, or in instances where it does, I have obtained permission to use such material and allow it to be made accessible worldwide via the Internet.

Signature: [redacted]

Vienna, Austria, June 2023 Thomas Kotrba

Acknowledgment

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and the Christian Doppler Research Association is gratefully acknowledged.

The computational results presented have been achieved in part using the Vienna Scientific Cluster [1].

[1]https://vsc.ac.at

Contents

List of Tables

List of Figures

Acronyms

ADAS Advanced Driver-Assistance Systems.

ANN Artificial Neural Network.

AP Average Precision.

BN Batch Normalization.

CNN Convolutional Neural Network.

DLA Deep Learning Accelerator.

DNN Deep Neural Network.

DVFS Dynamic Voltage and Frequency Scaling.

FN False Negative.

FP False Positive.

FPPI False Positives per Image.

IoU Intersection over Union.

L4T Linux for Tegra.

LAMR Log-Average Miss Rate.

LWIR Long-Wavelength Infrared.

mAP Mean Average Precision.

MLP Multilayer Perceptron.

MR Miss Rate.

MWIR Mid-Wavelength Infrared.

NIN Network in Network.

NN Neural Network.

PAN Path Aggregation Network.

ReLU	Rectified Linear Unit.
RoI	Region of Interest.
SGD	Stochastic Gradient Descent.
SoC	System on Chip.
SoM	System on Module.
SOTA	State of the Art.
SPP	Spatial Pyramid Pooling.
SWIR	Short-Wavelength Infrared.
TLU	Threshold Logic Unit.
TN	True Negative.
TP	True Positive.
VSC	Vienna Scientific Cluster.

Chapter 1

Introduction

In 2006 Hinton et al. [4] introduced a method to train a neural network to recognize handwritten digits (a task that was considered impossible at the time). They called this new method *deep learning*. This work marked the starting point for most research in machine learning we know today.

The field of visual object detection using Deep Neural Networks (DNNs) has become a heavily researched topic in the last decade. This development was driven by breakthroughs like the introduction of novel DNN architectures such as AlexNet[5] in 2012 or YOLO[6] in 2016, as well as the increasing demand for new use cases from the industry. They have realized the potential of deep learning models to solve tasks like autonomous driving and natural language processing, which are not practically applicable with conventional approaches. Many of the use cases are safety-relevant, like autonomous driving. When using a DNN based system as a critical component of driving assistance or a self-driving system, the error rate and, therefore, the quality of the object detection of the DNN becomes a top priority on the requirements.

1.1 Motivation

To a high degree, the error rate of an object detection system depends on the data the model sees during training and how similar the real-world input is compared to the training data. Training with larger datasets or adding image augmentation increases the size of the training set, therefore, can improve the trained network. When it comes to similarity with training data, there are often situations in the real-world scene that were not part of the training data. For example, new objects or surroundings, objects in never seen positions, angles or constellations, and often environmental changes in the scene like low illumination, lens glares, object occlusions, or adverse weather conditions.

In an object detection system with an image sensor sensitive to the visible spectrum of light, the information of the scene and the objects can get covered up or even lost by these latter-mentioned environmental influences. Therefore the robustness against these situations can only be as good as the remaining information of the scene. An approach to improve this is to use additional image sensors sensitive to different light spectra, which are not simultaneously affected by the influences in the same amount. The various sensors should deliver complementary information to the object detection system, making it more robust against environmental impacts and better overall.

A Convolutional Neural Network (CNN) for object detection consists of two main parts: a feature extractor (also called encoder), which compresses the original image and extracts features, and the head, where the regression of the bounding boxes of the recognized objects happens.

A multispectral object detection system can process the information using three different approaches:

- Run separate networks for each spectrum in parallel and fuse the results
- Fuse the multiple spectra images at the beginning
- Fuse the information inside the network

Regarding fusion inside neural networks, there are also different fusion schemes. Such a fusion network typically has multiple separate inputs and network paths for the different spectra until the graph reaches the fusion point. From there on, only one combined network graph exists. This type of network needs fewer computational resources than entirely separate networks because the multispectral data is processed in a single network branch after the fusion. When dealing with resource-constraint embedded hardware such as the NVIDIA Jetson platform, this becomes a key consideration in evaluating the practicality or workability of a specific approach.

1.2 Scope

The field of multispectral feature fusion in neural networks became very active, with ongoing improvements of State of the Art (SOTA) approaches. However, these novel fusion methods often exploit specific properties of the dataset to optimize the training. It is often unclear how these approaches would perform in a real use-case scenario and how well these often complex architectures would perform on embedded hardware regarding latency and power.

This thesis investigates the applicability of more general low-level multispectral feature fusion schemes in the YOLO4[7] architecture, which acts as a representative of a SOTA one-stage object detector. It evaluates the fusion architecture's impact on object detection quality and system performance on an

NVIDIA Jetson Xavier device. All measurements in this work are done with the KAIST multispectral dataset[8], a visible + Long-Wavelength Infrared (LWIR) pedestrian dataset. This work does not aim at improving SOTA performance. Instead, it examines the impact of more general fusion approaches with minimal design effort in application viability on an embedded device.

The work shall answer the following questions: How good is the object detection performance of general multispectral fusion architectures in YOLO4 compared to separate single-spectrum networks? What is the power consumption and network latency impact on an NVIDIA Jetson Xavier device? Is it beneficial in terms of applicability to use a fusion architecture over multiple networks?

To answer these questions, this work includes the following steps:

- Train separate networks for each spectrum
- Select and train several low-level multispectral fusion architectures
- Evaluate all networks by object detection and system performance metrics on an NVIDIA Jetson Xavier device
- Practical applicability analysis

The first step establishes a baseline for all metrics for the original network architecture and dataset. The initial network is trained for all input spectra separately. These trained networks act as references for the following fusion architectures. Furthermore, the weights from the trained networks are used as initial weights as a starting point for the training in the next step. Step two identifies basic low-level fusion architectures in the selected neural network architecture. These networks are trained with unified multispectral input images. Step three evaluates all networks from steps one and two regarding typical object detection performance and system performance metrics gathered from running these networks on the NVIDIA Jetson AGX Xavier. The last step is a discussion of what the results from step three mean for a real application scenario.

1.3 Contributions

To summarize the contributions of this thesis:

- Six multispectral fusion approaches in the CSPDarknet53 backbone of YOLOv4 are proposed and trained with little design effort
- The fusion architectures are evaluated on the NVIDIA Jetson AGX Xavier, by comparing object detection performance and system-specific performance metrics, such as network latency and power consumption.

1.4 Outline

Chapter 2 describes background knowledge and SOTA about object detection, architecture- and training of DNNs, and sensor fusion in neural networks. Chapter 3 gives a detailed overview of how the experiments are set up. Chapter 4 explains the selected low-level fusion approaches in the YOLO4 architecture and their key points. The experiments themself and discussions on the results are documented in Chapter 5. Finally, chapter 6 concludes all experiments carried out in this thesis.

Chapter 2

Theory and State of the Art

2.1 Object Detection

The task of object detection is defined as recognizing and localizing objects in a scene. The resulting information is the type and location of objects in a given scene. The input for such an object detection system should represent the real environment. In the case of visual object detection, an image recorded in the visible spectrum acts as the input. The information on the detected objects is represented as class labels and bounding box coordinates. A class label contains the object's type, and bounding box coordinates describe a rectangle fully enclosing the object. In addition, a confidence value is expected to accompany a detection, indicating how confident the detector is about its prediction.

2.1.1 Metrics

Several metrics exist to compare the results of different object detectors. The most often used metric is the Mean Average Precision (mAP). A less frequently used is the Log-Average Miss Rate (LAMR). Before these two are reviewed in detail, it is defined whether a prediction of a detected object is correct or incorrect.

There are two requirements for a detection to be considered correct: 1.) The class label has to match the real object. 2.) The predicted bounding box must match the correct one (also called ground truth) to a certain amount. Detecting the position of an object is a regression task; therefore, the detected box will never match the real object precisely. As a metric on how well the predicted bounding box matches the ground truth, the Intersection over Union (IoU) is used. The IoU describes the ratio of the intersecting area over the unified area between the detected and the ground truth bounding boxes with

Equation 2.1.

$$\mathrm{IoU} = \frac{I}{U} = \frac{A_{\mathrm{predict}} \cap A_{\mathrm{gt}}}{A_{\mathrm{predict}} \cup A_{\mathrm{gt}}} \tag{2.1}$$

For evaluation, a threshold is defined and compared to the IoU value of the detection: If the calculated IoU is above the threshold, the detection counts as correct; if it is below the threshold, the detection counts as incorrect. The metric has some disadvantages. For example, its value drops to 0 as soon as there is no more overlap. Several attempts were made to improve the properties of the IoU to make it usable as a loss function in machine learning training. Examples are GIoU [9] and DIoU [10].

According to the chosen IoU threshold, each detected (and not detected) object can be categorized into one of the following three types:

- True Positive (TP): correct detection of the ground truth (IoU was higher than threshold)
- False Positive (FP): incorrect detection of the ground truth (IoU was below the threshold) or detection of a non-existing ground truth object (IoU was 0)
- False Negative (FN): a ground truth object with no assigned TP

Notice here that sometimes the concept of a True Negative (TN) is used. TNs are all (background) objects that are correctly not detected as objects. Because this is not a helpful calculation category, it is often ignored. An illustration of these described three categories is shown in Figure 2.1. Applying this category assignment to all objects of a test set of multiple images delivers total numbers for each category. These are the base for more comprehensive metric calculations.

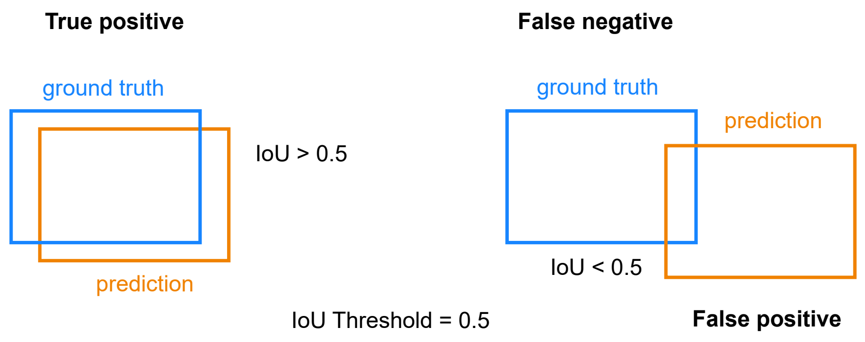


Figure 2.1: Illustration of TP, FP and FN.

Mean Averagae Precision (mAP)

The Mean Average Precision (mAP) is based on the precision-recall (PR) curve. This curve represents the relationship between the precision P and the recall R, which are defined as

$$P = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \quad \text{and} \quad R = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{2.2}$$

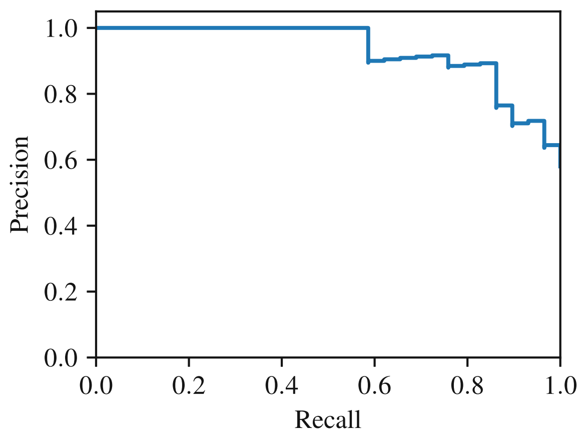


Figure 2.2: Precision - Recall curve example.

The precision represents how many of the detections made are correct, and recall describes how many of the ground truth objects are detected. Both values are dependent on the IoU threshold through TP, FP, and FN.

First, all detections have to be assigned to either TP, FP, or FN according to the chosen IoU threshold. A confidence threshold is introduced for drawing the curve: It starts at 1 and continuously decreases until it reaches 0. For every step, there will be a specific number of TP, FP, and FN. Every time one of these numbers changes, precision and recall are recalculated, and the resulting point is drawn into the curve. The resulting curve looks similar to the one shown in Figure 2.2. The Average Precision (AP) can be calculated based on that curve. The AP is a metric that describes the precision-recall curve with a single number. Interpolation is done before the actual calculation to reduce the effect of the "wiggles" in the curve (as seen in Figure 2.2) [11]. In the literature, there are two calculation methods: The 11-point interpolation and the all-point interpolation method.

In the 11-point method, the AP is calculated by setting the precision at 11 evenly spaced recall levels $\{0, 0.1, 0.2, ..., 1\}$ to the maximum precision obtained for any recall $R' \geq R$, followed by averaging these gathered 11 precision points with

$$\text{AP}_{11} = \frac{1}{11} \sum_{R \in \{0,0.1,0.2,...,1\}} P_{\text{interp.}}(R) \tag{2.3}$$

where

$$P_{\text{interp.}}(R) = \max\big(P(\tilde{R})\big), \quad \text{with } \tilde{R} \in \big[R, R_{\max}\big] \tag{2.4}$$

The 11-point calculation method was the default method for the object detection task in the PASCAL VOC challenge until 2009. Because it was too coarse when comparing low AP scores, it was replaced with the all-point calculation method for the contest in 2010 and later [11].

In the all-point method, every available precision value is used for calculation. The interpolation of the curve works with the same mechanism as in the 11-point method. Conceptual this is the same as calculating the area under the interpolated curve.

$$\text{AP}_{\text{all}} = \sum_{n=0}^{N} (R_{n+1} - R_n) P_{\text{interp.}}(R_{n+1}) \tag{2.5}$$

where

$$P_{\text{interp.}}(R_{n+1}) = \max\big(P(R_m)\big), \quad \text{with } n+1 \leq m \leq N \tag{2.6}$$

While the AP is typically calculated for each object class, the mAP combines the APs by averaging over the APs of all classes with

$$\text{mAP} = \frac{1}{K} \sum_{k=0}^{K} \text{AP}_k \tag{2.7}$$

where K is the number of classes in the dataset. Because the mAP and AP values need the information of the IoU threshold to be comparable, the threshold value is often carried as a subscript, e.g., $\text{mAP}_{.50}$ and $\text{AP}_{.50}$ for a threshold of 0.5.

Log-Average Miss Rate (LAMR)

The base of the Log-Average Miss Rate (LAMR) is the curve of Miss Rate (MR) over False Positives per Image (FPPI). These two metrics are directly related to precision and recall over

$$\text{MR} = 1 - R \quad , \quad \text{FFPI} = 1 - P \tag{2.8}$$

While the precision-recall curve focuses on correctly detected objects, the miss rate-FPPI curve focuses on missed detections. Miss rate curves are sometimes preferred over the precision-recall curves because, in certain applications like automotive, there is often an upper limit to FPPI, independent of the total number of objects to detect [12, 13]. To summarize the curve in a single metric, the Log-Average Miss Rate (LAMR) is used. It is calculated by averaging the miss rate at nine FPPI values evenly spaced in logarithmic space from 10^{-2} to 10^{0} [12], that is

$$\text{LAMR} = \frac{1}{9} \sum_{k=0}^{8} \text{MR}\big(x(k)\big), \quad \text{with} \quad x(k) = 10^{\frac{k}{4}-2} \tag{2.9}$$

2.2 Neural Networks

Neural Networks (NNs) plays an important role in modern machine learning. They form the backbone of deep learning, a subset of machine learning. Neural networks are the core of several everyday tasks like surveillance, Advanced Driver-Assistance Systems (ADAS), speech recognition, and many more. This section lists the most important concepts of neural networks and how they are trained.

2.2.1 Basics

Artifical Neurons

A Neural Network (NN) (sometimes called Artificial Neural Network (ANN) for better differentiation) is a structure that is inspired by the structure of the neurons in the human brain. A mathematical model of how the neurons in the brain might interact with each other was first described by McCulloch and Pitts [14] back in 1943. Their proposed model was pretty simple by providing multiple binary inputs and one binary output port: The output gives a logical 1 if a certain number of inputs are 1. Despite the simplicity, the authors showed that combining multiple neurons to construct logical operators is possible.

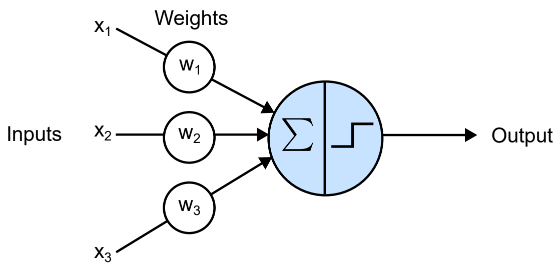


Figure 2.3: Illustration of a TLU.

In 1958 Rosenblatt [15] laid out the fundamentals for modern NNs with his work about the *perceptron* model. In literature, Rosenblatt is sometimes referred to as the “father of deep learning”. In contrast to the artificial neuron from McCulloch and Pitts, the inputs and outputs of the basic building block in this model, the Threshold Logic Unit (TLU) (Fig. 2.3), are represented by numbers. In addition, every input has an assigned weight. To calculate the output of a TLU, first, all input numbers are multiplied with their according weights and summed up with

$$s = \sum_{n=1}^{N} x_n w_n = \boldsymbol{x}\boldsymbol{w}^\top \tag{2.10}$$

A threshold function H is then applied to the calculated sum s, resulting in the output y. In the simplest case, this is the Heaviside step function. The threshold can be adjusted away from 0 with a bias term.

It is realized by adding an input node, called "bias node", which is constant 1. Therefore, the weight for this specific node is always a part of the sum s. The bias weight is separated from the weights of the regular inputs and written as an extra term b so that the output can be written as

$$y = H\Big(\sum_{n=1}^{N} x_n w_n + b\Big) = H\big(\boldsymbol{x}\boldsymbol{w}^\top + b\big) \quad \text{, with} \quad H(x) := \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases} \tag{2.11}$$

To this day, the TLU is the basic building block of most NNs. A single TLU can be practically used for binary classification tasks. The decision boundary is a hyperplane on the input space.

The Perceptron

Multiple TLUs can be combined to form a layer. Every input (including the bias) is represented via a node in the input layer. Each node is connected to every TLU. This so-formed network (Fig. 2.4) is known as the *perceptron*[15] - a fundamental architecture of a NN. It became the first model of a NN that could learn the necessary weights for a specific classification task by providing input samples [16]. A layer where every input is connected to every TLU of the layer is also known as *fully connected* or *dense* layer. The perceptron shown in Figure 2.4 could be used for multi-label or multi-class classification tasks.

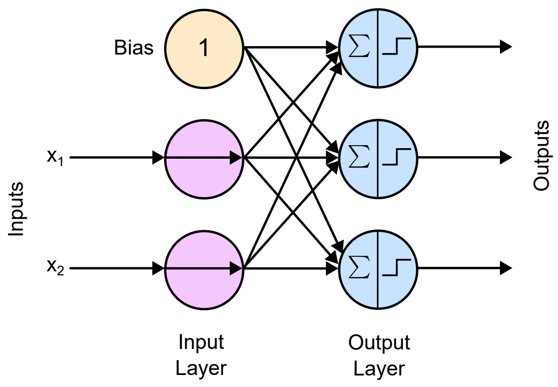


Figure 2.4: Illustration of the Perceptron.

Analog to that, networks with multiple fully connected layers can be built (Fig. 2.5). This network structure is known in the literature as Multilayer Perceptron (MLP). Layers without direct connection to the input or the output nodes are called *hidden layers* because they are not visible to the outside. Such networks are typically referred to as Deep Neural Networks (DNNs). However, in the literature, there are different opinions on if a NN with a single hidden layer can already be considered "deep" because modern NNs architectures can be several hundred or thousand layers deep. Compared to the perceptron, the Multilayer Perceptron (MLP) can solve more complex tasks, like the XOR-problem, which was pointed out in a work of Minsky and Papert [17].

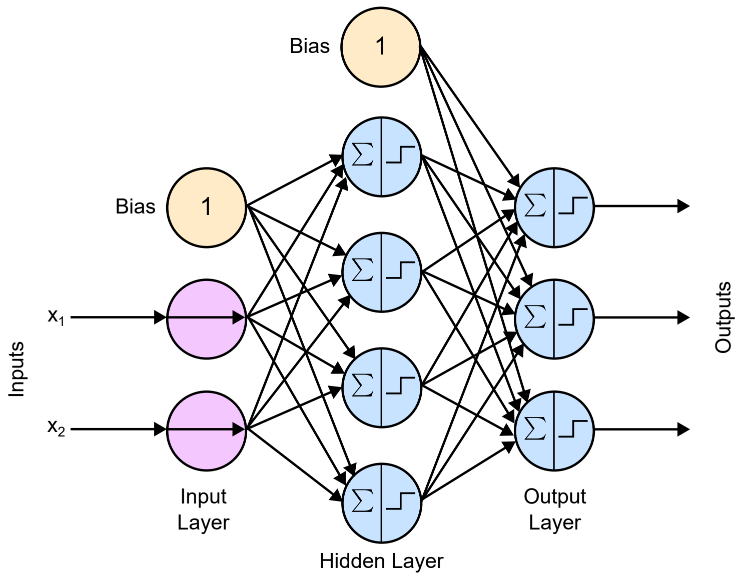


Figure 2.5: Illustration of an MLP.

2.2.2 Training

Training a NN refers to the task of finding values for weights and biases in a network to make it behave in an intended way. For example, in a classifier network, this would mean setting the network parameters so that any given input data point is classified correctly.

The training of a perceptron, like the one shown in Figure 2.4, is a reasonably straightforward task. The learning rule for this type of network was influenced by the work of Hebb [18] and later became known as “Hebb’s Rule”. The rule states that when neurons often interact, this connection is strengthened and therefore becomes more efficient.

When training a Perceptron, an output is generated for each node for every sample in training. The error between the target output of each node and the produced output determines the amount the weight for this particular connection changes to make it more likely that the output becomes correct. The change of the connection weight between an input node i and the output node j between two consecutive iterations is determined by

$$\Delta w_{i,j} = x_i(y_j - \hat{y}_j)\eta \tag{2.12}$$

, where y_j is the target output of node j, $\hat{y}_j$ is the actual output in the current iteration, x_i is the current input i, and η is a weighting factor called *learning rate*.

Training of a MLP, like the one shown in Figure 2.5, and more complex networks require a more sophisticated calculation scheme, which became available in the 1980s.

Back-Propagation

The algorithm of back-propagation was independently described and mentioned by several researchers during the late 1960s to the 1980s. It had its breakthrough in popularity with the work of Rumelhart et al. [19] in 1986. (Goodfellow et al. [16] give a comprehensive historical background summary). It became popular in machine learning and is still a common and well-understood learning technique for neural networks. This section should give a brief overview of what the algorithm does and how it is used for training a neural network without going into complex mathematical expressions.

The term back-propagation comes from the principle of operation of the algorithm, which does a backward propagation of errors through the network. A common misconception is that back-propagation describes the whole training process of a neural network, including gradient descent, while it technically only describes the calculation of gradients [16].

How well the output of a neural network matches the desired outcome (target) is expressed with a loss (or cost) function (Sec. 2.2.2). This function depends on all network parameters, such as weights and biases. To adjust these parameters to improve the network's output, it is necessary to know how much each of them influences the loss of the network. To determine this influence, the loss function is derived with respect to all network parameters. While this can be easily expressed analytically, it is challenging to calculate it numerically efficiently.

The back-propagation algorithm uses the chain rule to calculate the derivatives of the network's output with respect to the parameters and inputs of the last layer. It now knows how the error was influenced by the layer parameters and inputs (the outputs of the layer below). The algorithm then descends into the layer below and repeats the same calculations. It does this recursively for all layers below until the neural network inputs are reached. Performing this from the outputs to the network's inputs means that the error from the loss function is propagated backward through the network - hence the name back-propagation.

On a high-level basis, the concept of training a neural network using back-propagation looks like the following:

1. Generating outputs by passing input information (training instances) through all network layers. Starting at the first layer of the network, each layer produces an output, which consecutively is used as input for the following. All intermediate results are stored. This is called “forward pass” or “forward propagation”.

2. Measuring how well the output of the last network layer matches the desired output. The error is usually calculated with a loss or cost function (Sec. 2.2.2).
3. Calculating the influence each network parameter has on the loss. Partially derivating the loss function of the network with respect to all parameters by calculating it recursively by using the chain rule. This is the actual scope of the back-propagation algorithm.
4. Performing an update of the network parameters using the calculated gradients to do a gradient descent step.

This described process is repeated until the solution converges or is stopped. How often the parameter updates happen depends on the used variant for gradient descent. In a "batch gradient descent", all training samples' gradients are used to update the weights. In a "stochastic gradient descent" the weights are updated after each training sample. A compromise between these two methods is the "mini-batch gradient descent", where the weight updates happen after a specific number of training samples. The whole set of all training examples is called an "epoch". Training is usually run for several epochs so that every sample of the training set is used multiple times.

However, an MLP (like the one in Figure 2.5) can not be trained with back-propagation without changes. The reason is that the calculated gradients for the network parameters determine the direction of the consecutive weight update. Hence, by using a threshold function like the Heaviside function, one runs into the problem that all segments are flat, leading to zero gradients. To show the capability of training an MLP, Rumelhart et al. [19] changed the activation function to the sigmoid function. It was considered a good choice for a long time because research found that biological neurons implement an activation function similar to it [20]. Later it turned out that other activation functions provide better performance for DNNs [21].

Activation Functions

Activation functions play a crucial role in DNN architectures. They are the reason, why a DNN is capable of solving more complex tasks than single layers: they create non-linear separations between the layers. If a linear function is used as an activation function, all layers could be simplified to just one linear transformation. Hence it would make the additional layer useless because the capability of learning a task would be restricted to the capacity of a single layer. Some of the most popular ones are shown in Figure 2.6.

Activation functions can be categorized into saturating and non-saturating functions. The category is important when training the DNN. Saturating functions like sigmoid get flat when the input becomes either strongly positive or negative, respectively, while the derivative becomes very small. If the layers'

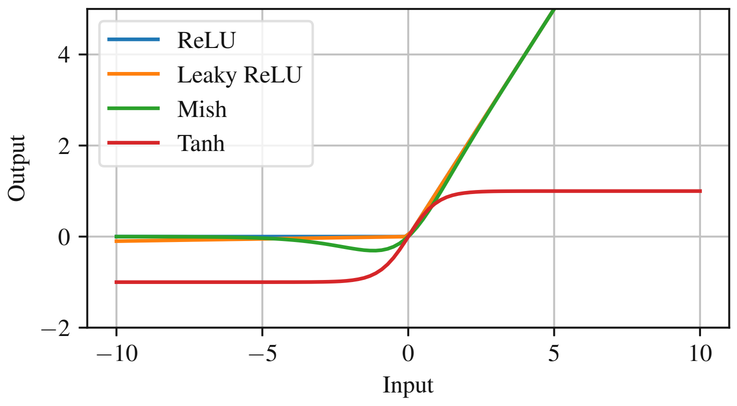


Figure 2.6: Common activation functions.

weights are initialized unfavorably large (positive or negative), it is easy for the output of the activation function to get into the saturation areas. During the back-propagation step in training, the already small gradient becomes more diluted while approaching the network's input. This effect is called the vanishing gradient problem. The consequence is that weight updates have more impact on the later than on earlier layers. This makes it hard for a DNN to converge to a solution. In contrast, it is also possible that the gradients become more significant from one layer to another. The described effect is known as the exploding gradient problem.

Using non-saturating activation functions is one key for effectively training DNNs. The most commonly used one is the Rectified Linear Unit (ReLU) [22], proposed by [5, 21]. It performs better when training DNNs than the sigmoid function [21]. The ReLU is non-saturating if the input is positive. However, if the input becomes negative, it outputs a constant zero. Because the gradient is also zero, it is hard to get out of there again, and the ReLU can "die"'. This behavior is known as the "dying ReLU" or "dead ReLU" problem [23].

A way to solve this is to use a non-saturating ReLU. A popular choice is the Leaky ReLU, a version with a slope assigned to negative values. Other popular activation functions are Exponential Linear Unit (ELU), Self-normalizing ELU (SELU), Sigmoid Linear Unit (SiLU), also called Swish, and Gaussian Error Linear Unit (GELU).

Normalization

Normalization in the context of DNNs describes standardizing the inputs of layers in the network. It tackles the problem of "internal covariate shift" which describes changing distribution statistics in the activations of a layer caused by the changing network parameters during the training [22]. Internal covariate shift can cause unstable gradients during training of DNNs and prevent the training from converging.

The most popular normalization method is Batch Normalization (BN), proposed by Ioffe and Szegedy [24] in 2015. It is often implemented as an additional layer, which gets added to the network before or after the activation layer. The BN ensures that each layer can adjust its inputs to the optimal scale and mean during training. First, the inputs are standardized (zero mean and variance of one). This happens by calculating the mean and variance for each input of the layer over the current mini-batch during the training (Eq. 2.13 and 2.14). After that, the inputs are scaled and shifted with two parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$, which are trained in the back-propagation process via gradient descent.

$$\boldsymbol{\mu}_B = \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{x}_i \tag{2.13}$$

$$\boldsymbol{\sigma}_B = \sqrt{\frac{1}{m}\sum_{i=1}^{m} (\boldsymbol{x}_i - \boldsymbol{\mu}_B)^2} \tag{2.14}$$

$$\hat{\boldsymbol{x}}_i = \frac{\boldsymbol{x}_i - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B{}^2 + \epsilon}} \tag{2.15}$$

$$\boldsymbol{y}_i = \boldsymbol{\gamma} \odot \hat{\boldsymbol{x}}_i + \boldsymbol{\beta} \tag{2.16}$$

where: $\boldsymbol{\mu}_B$ contains the mean over the mini-batch for each input element, $\boldsymbol{\sigma}_B$ contains the standard deviation over the mini-batch for each input element, $\boldsymbol{x}_i$ contains all input elements for the current mini-batch i, $\hat{\boldsymbol{x}}_i$ is the standardized input with zero mean and standard deviation of one, and $\boldsymbol{y}_i$ is the scaled and shifted output of the BN layer.

Noticeably, BN behaves differently during training and test operations. While training means and deviation can be calculated from the mini-batch, in testing, the mini-batch might be too small to deliver reliable numbers. Usually, either moving averages for mean and deviation are generated during training[20], or they are generated after training by running the complete training set through the network [20, 22]. These parameters are fixed and then used during the test phase. For speedups during the test phase, it is often possible to combine the BN layer with the previous layer [20, 22]. This combining is done by merging the transformation, which would be performed by the BN layer, directly into the previous layer. This combination is called a fused BN layer.

At first glance, it looks like BN increases the time it takes for the training to converge by adding many extra calculations. However, the more stable gradients enabled through BN allow the training to be much more robust, which allows higher learning rates, ultimately leading to a faster-converging training [20, 22]. Ioffe and Szegedy even showed, that by using BN, typically disadventagious activation functions like sigmoid can be used.

Residual Connections

As proposed by He et al. [1], Residual connections are another method to tackle the vanishing gradient problem in DNNs. With their proposed approach, they were able to train deeper networks, which allowed them to win several challenges.

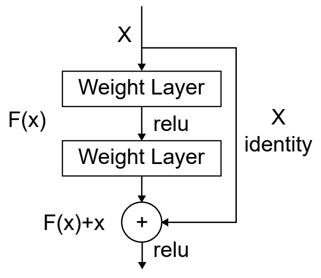


Figure 2.7: Residual block[1].

Initially, the authors observed a saturation in the training accuracy of their studied DNNs when they increased the network depth (i.e., the number of layers). They concluded that it becomes too difficult for the training to map the wanted function onto the stack of nonlinear layers at a certain point. Furthermore, the authors observed that a solution of these deep DNNs could be constructed by hand when they take a shallower network and add additional layers, which map the identity function. The resulting DNN turned out to have the same accuracy as the shallow one. To exploit this, they changed small blocks of layers to map the residual $F(x) := H(x) - x$ instead of the originally desired mapping $H(x)$. The original mapping is recasted to $H(x) = F(x) + x$. The x term is realized in the network by adding a skip connection (Fig. 2.7).

Even though it should be equally possible to map a stack of nonlinear layers to the initially desired mapping H as to the residual $H(x) - x$, it is easier for the training to do the latter. The skip connections allow the gradients to flow freely through the network, enabling the training of the whole network, even if some parts have not started adapting [20, 22].

Loss Function

The loss function or cost function plays an essential role in the training process (see Sec. 2.2.2) of DNNs. It describes how well the current output of the network matches the desired one. Which specific loss function is used is highly dependent on the actual task of the network. I.e., the loss function of a DNN for linear regression will almost certainly look different than the loss of an object detection network. However, loss functions can be loosely categorized into regression and classification loss functions.

Regression loss functions are used for tasks where the network should predict an actual output value. In contrast, classification loss functions are meant for tasks where the output consists of predicted probabilities of predefined categories. Commonly used loss functions are Mean square error (MSE), Mean absolute error (MAE), Hinge Loss, Binary Cross-Entropy, and Categorical Cross-Entropy Loss.

While loss functions for simple tasks, i.e., regression of an output value, are usually primitive, they can become complex for tasks where multiple regression values and labels need to be predicted. For example, in object detection, there are usually multiple losses defined: one for the bounding box regression and one for the label classification. These are then combined, e.g., with a weighted average, to form the overall loss function of the object detection network.

Two typical loss functions used in object detection are MSE for bounding box regression and Cross-Entropy loss for classification. The MSE between the ground truth bounding box A and a predicted bounding box B is calculated with

$$\mathrm{MSE}\,(A,B) = \frac{(A_{x_1}-B_{x_1})^2+(A_{y_1}-B_{y_1})^2+(A_{x_2}-B_{x_2})^2+(A_{y_2}-B_{y_2})^2}{4} \tag{2.17}$$

The Cross-Entropy loss between the ground truth class probability p and the predicted probability q is calculated via

$$\mathrm{CE}\,(p,q) = -p * \log(q) \tag{2.18}$$

Overfitting and Regularization

Regularization techniques help to prevent the network from overfitting. Overfitting occurs when the network works better on the training data than on the validation data. Often this is caused when the network is unnecessarily complex (i.e., more network parameters) for the given task. It tries to fit the training samples as accurately as possible during training. Because the training data is never perfect, it contains noise from measurement errors, sampling noise caused by too few data points, or other reasons. The network tries to model this noise as part of the training data with some high-degree polynomial functions [20]. Figure 2.8 shows a simple example of overfitting.

The state of a network to fit the training samples is also referred to as "capacity". In Figure 2.8a, the model's capacity is too low to represent the actual function, referred to as underfitting. Figure 2.8b has the suitable capacity to model the original function properly. The model in Figure 2.8c has a too high capacity (allows polynomials to 15th degree), which causes it to overfit the data. That is because the more complex function is better at modeling the noise from the data points. Even though this function fits the sample points the best, the MSE on the validation set shows that it will not generalize well.

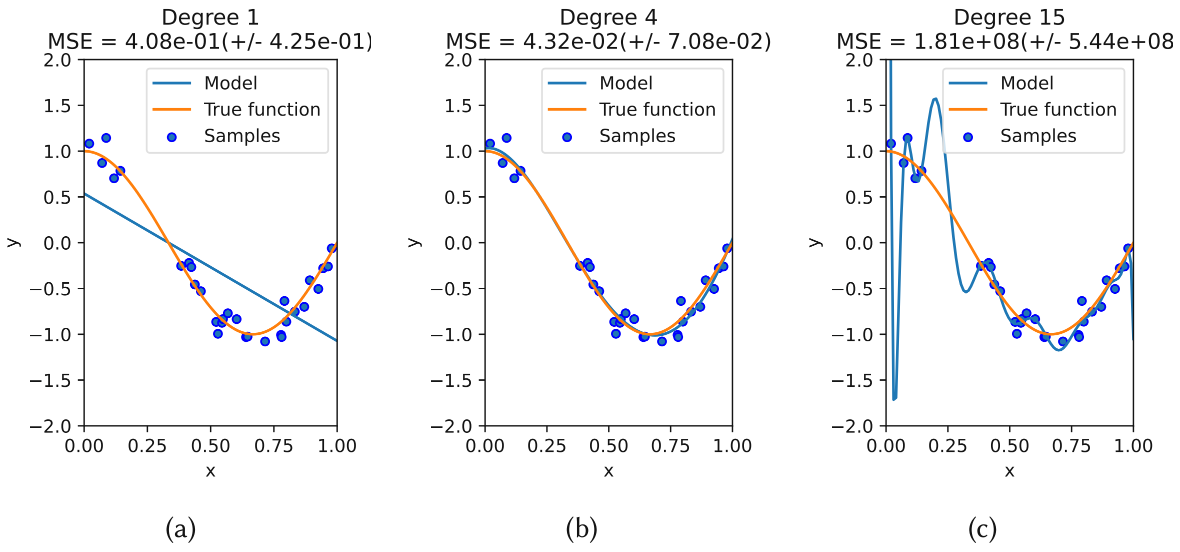


Figure 2.8: Examples of different function fits.
(a) Underfitting; (b) Optimal Fit; (b) Overfitting; Code from [2].

To prevent a model from overfitting, commonly used approaches are:

- Reduce noise in training data by improving sample quality or by adding more samples
- Simplify the model either by changing the network itself or by constraining, e.g., the network parameters

While one often has limited control over the quality of the training samples, the model can be more easily constrained during the training. Some popular methods are outlined below.

The network parameters are continuously adjusted during the training phase to fit the data. The network will go from a state where its capacity is too low for the desired function (Fig. 2.8a) to a state with a good fit 2.8b), and finally to overfitting 2.8c. Stopping the training before it reaches the overfitting state is called "early stopping". To find the optimal stopping point, the loss is evaluated on a set of data the network has never seen during training, e.g., a test set. This metric is referred to as generalization loss. While the training loss decreases alongside the generalization loss, there is a point at which the generalization loss starts to increase again. This is the optimal stopping point. Figure 2.9 shows an example of early stopping. The left side of the red dotted line shows the area where increasing network capacity results in lower training and generalization loss (underfitting). The generalization error is represented here through the validation loss as an approximation. After reaching the optimal capacity, the generalization error increases with additional capacity (overfitting).

Another often-used method for regularization is to constrain the model capacity via the size of the parameters. This is done by adding a regularization term to the overall loss function to penalize the pa-

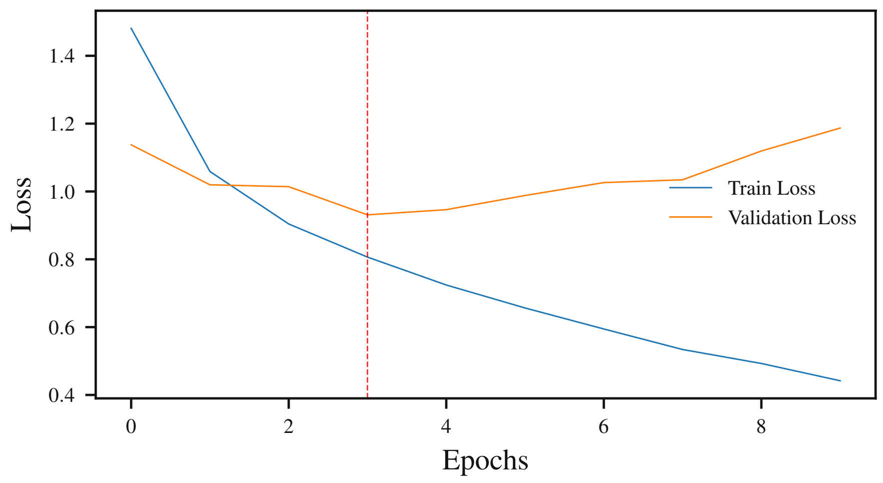


Figure 2.9: Example of early stopping.

rameters. Two norms are commonly used to calculate the penalty: L1 (also known as Lasso Regression) and L2 (also known as weight decay, Ridge regression, or Tikhonov regularization [16, 25]).

Assume the loss function of a given network is

$$L = Loss\left(y, \hat{y}\right) \tag{2.19}$$

Adding the absolute sum of the weight vector forms the L1 regularization

$$L = Loss\left(y, \hat{y}\right) + \lambda \sum_{i=1}^{N} |w_i| \tag{2.20}$$

, while adding the squared sum of the weight vector forms the L2 regularization.

$$L = Loss\left(y, \hat{y}\right) + \lambda \sum_{i=1}^{N} w_i^2 \tag{2.21}$$

, where λ is the factor of the regularization term.

In both cases, the additional term causes the training to minimize the sum of the new loss function. Practically this means the parameters themself become smaller than without regularization. λ controls the effect of the weight shrinkage [16]. A larger λ results in smaller weights, leading to a better generalization.

An essential difference between the two methods is that L2 tends to shrink all weights to small values, while L1 tends to bring them to zero. This effect can be visually illustrated with an example. Assuming a simple model with only two parameters, the regularization term could be expanded to $L_1 = |w_1| + |w_2|$

and $L_2 = w_1^2 + w_2^2$, respectively. Constraining both equations to a value t results in a diamond (Fig. 2.10a) and a circle-shaped (Fig. 2.10b) region when drawn onto a two-dimensional plane. Also, a line is drawn, representing the original loss function. During training, the parameters will settle on the contact points between these contours. The size of the constraint area can be steered by λ.

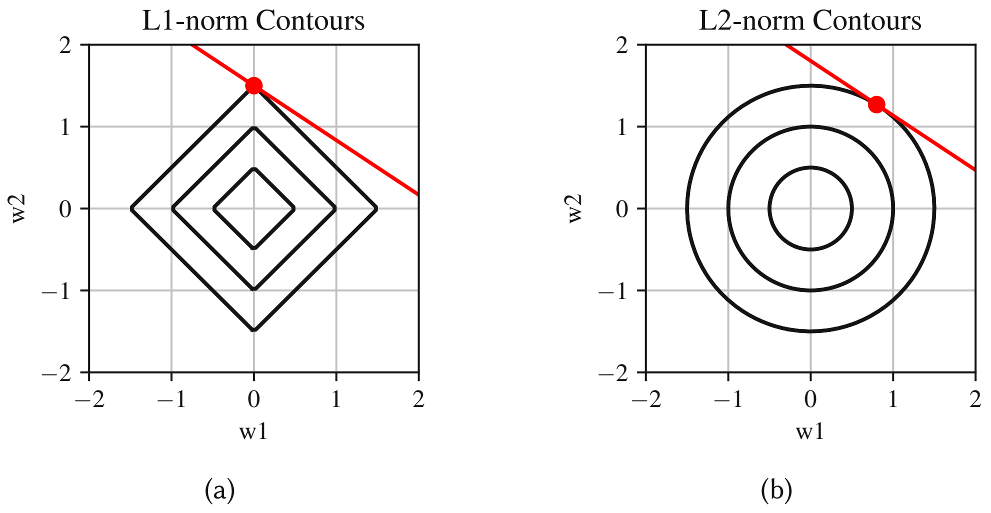


Figure 2.10: Visualization of L1 and L2 regularization. (a) L1 regularization; (b) L2 regularization.

When t gets smaller, due to the diamond shape of the L1 constraint area, parameters tend to be pushed to an axis, which makes one parameter zero. For L2, the parameters also get smaller, but not to precisely zero.

Optimizer

The weights are usually updated with gradient descent when training a neural network. The gradients are calculated for every step, and the weights are adjusted to the steepest descent direction. This is not always the fastest way to get to the global minimum. While the batch gradient descent method uses all images from the training dataset to calculate the gradients and update the parameters, Stochastic Gradient Descent (SGD) typically uses mini-batches. This can lead to faster-converging training but comes at the cost of introduced noise due to the random sampling [16]. In contrast to batch gradient descent, this noise does not go away in a local minimum, so the learning rate has to be reduced during training [16]. Several optimizer methods exist which try to improve the speed and direction of the current gradient descent step. They do this, for example, by including old gradients in the current calculation or by calculating the direction a bit ahead.

Frequently used optimizers are: Momentum[26], Nesterov Momentum[27], AdaGrad[28], RMSProp[29], and Adam[30]. Momentum is an extension to SGD. It adds an additional term that allows the accumulation of a moving average of past gradients. Nesterov Momentum is a variant of Momentum that was inspired by Nesterov's gradient descent [31]. The difference to Momentum is that the gradient is evaluated after the current velocity is applied [16]. AdaGrad and RMSProp try to adjust the learning rates for all parameters regarding their historical values [16]. Adam, simply spoken, combines the advantages of RMSProp and Momentum [16].

Learning Rate Sheduler

Finding the optimal learning rate is another crucial part of a network converging. If the chosen learning rate is too low, converging will take longer. If the learning rate is too high, the network converges to a suboptimal solution [20]. While finding the optimal learning rate is a complex and time-consuming task, continuously decreasing the learning rate as the progress slows down can make a network converge faster than with the optimal learning rate [20].

Several different approaches exist to reduce the learning rate during training. A simple approach is the step scheduler, which reduces the learning rate at specified epoch numbers in training. Other approaches are more sophisticated and follow exponential, sigmoid, or cosine functions. Some approaches also implement restarts like SGDR, introduced by Loshchilov and Hutter [32]. The restarts decrease the possibility that the training gets stuck in a local minimum.

2.2.3 Convolutional Neural Networks

Until now, it was not specified in more detail what the layers of a DNN may look like. At the beginning of this chapter, the fully connected layer was introduced. A fully connected layer is capable because each input is connected to every output. At the same time, this property makes a fully connected layer a resource-intensive layer because of the number of parameters a single layer has.

A convolutional layer uses a property inspired by nature: receptive fields in the visual cortex. Generally, a receptive field describes the number of inputs influencing a single output in a layer. Researchers found that the neurons in the eyes of living beings are structured this way: the whole perceptive field of the human eye is split into many smaller local receptive fields, overlaying each other for a small amount. Instead of every input influencing all outputs, in nature, only the output neurons are influenced, in which perceptive fields the input lies. In other words: every input can only have a local influence on the output. By stacking multiple layers, these receptive fields cascade and eventually increase in size through the layers [16]. This way, connections across the whole input layer are possible over several layers.

In a convolutional layer, the perceptive fields are realized with squares (e.g., 3×3) containing the

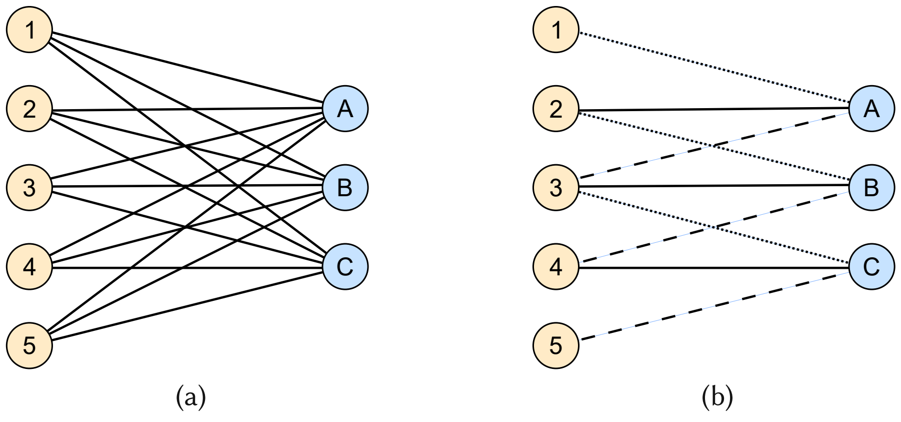


Figure 2.11: Comparison of: (a) fully connected layer; (b) convolutional layer.

weights for an area of this exact size. This square is referred to as a filter. To calculate the outputs, this filter is slid over the whole layer by a specific stepsize in the x and y direction (called stride), multiplying and summing up all weights with their underlying inputs in every step. The weights of the filter stay the same over the spatial area, thus leading to a significant reduction in network parameters. The name convolutional layer comes from the fact that this calculation scheme is very similar to the mathematical operation of convolution.

While there were some approaches of CNNs before, [33] was the first work using backpropagation to train these convolutional layers. The breakthrough for CNNs was arguably the introduction of LeNet-5 by Lecun et al. [34]. Banks widely used it for hand-written digit recognition on checks [20].

2.3 Object Detection using Convolutional Neural Networks

The fact that CNNs are inspired by how neurons are structured in the visual cortex makes them particularly suitable for processing images and, therefore, for computer vision tasks. Typical tasks are image classification, image segmentation, and object detection (briefly described in Section 2.1). This section presents the basic principles of how object detection with CNNs work.

2.3.1 Feature Extraction

For most computer vision tasks, a visual image of a scene marks the network's input. The properties of an image are essentially the spatial resolution, i.e., width and height in pixels, and the information each pixel represents, e.g., three channels with 8 bits. When dealing with images as inputs for DNNs, commonly, each pixel of the image is spatially aligned to one input node of the first layer. However, it is common to scale the image to a smaller size beforehand.

The first step for any computer vision CNN is interpreting the image. This task is called "feature extraction". Features are components of the image that characterize its content. These can be low-level properties like colors or structures and larger things like shapes or objects. What the extracted features represent depends on the task and the training and cannot be predicted. Ideally, the network learns to extract the essential features for the solution of the respective task.

To extract these features from an input image, computer vision tasks usually use CNNs because of their previously described properties to process images efficiently. The layers are typically a sequence of a convolutional layer, activation layer, and maxpool layer. The maxpool layer is used for scaling down the image in the spatial dimension while extracting more sophisticated high-level characteristics of the previous layer with deeper kernels (more channels). Analog to a maxpool layer, a spatial scale down can also be achieved by a convolutional layer with a stride value greater than one. Several of these described layer blocks are arranged and repeated in a specific pattern to form the so-called "backbone".

It is a common practice to use the backbone of (originally) classification networks for general-purpose feature extraction in other, more complex, network architectures such as object detectors and image segmentation. A notable backbone family is ResNet.

2.3.2 Bounding Box Regression and Classification

After the backbone, the now extracted features of the input image need to be further processed to gather information on what objects are present in the image and where those are located. Determining whether a given portion or region of the image contains an actual object is done by classifying the given area. A network for image classification typically consists of a feature extractor like ResNet50 followed by a classification head, e.g., an MLP. The output is a vector containing probabilities for all possible classes the network was trained to classify.

The more challenging part of object detection is identifying the regions that could contain an object. There are two main types of object detectors: two-stage and one-stage detectors. A significant, but not the only, difference is how they generate the region proposals. A short introduction to these working principles is given in the following section.

Two-Stage Detectors

The name of the two-stage detectors comes from the fact that they detect objects in two steps. First, the network tries to extract Regions of Interest (RoIs) from the image. These regions are then passed to a classification section, where each region is evaluated if it contains an actual object. A crucial step in this process is to find the RoIs efficiently.

Perhaps the most straightforward approach to finding suitable regions, as described by many authors, is the “sliding-window” approach. In this approach, a window, which can be varied in size and ratio, is slid over the input image for all possible combinations. Each cropped-out portion from the image is fed into a trained classification network, which outputs the class probabilities (including a background class for no object in the window). Arguably this approach is not feasible for efficient applications due to its need to repeatedly process all regions.

R-CNN [35], an early member of the region based CNNs, tackled the region proposal search by limiting it to roughly 2000 proposals that are generated by the selective search algorithm. Each of the 2000 regions is then cropped out of the input image, scaled to a fixed size, and individually fed through a CNN, which does not only classify but also regress correction coordinates for finer adjusting the predicted bounding box, from the region proposal stage. A significant downside is that it is computationally expensive because processing has to be repeated for each of the 2000 regions for a single image.

Fast R-CNN [36] improved the performance by feeding the whole input image only once through the feature extraction network. For the final classification and regression, only the extracted features from this stage are further processed. This significantly reduces the needed computation time compared to R-CNN because many computations are shared within the feature extraction stage.

Faster R-CNN [37] took it one step further and replaced the selective search algorithm for the region proposals with a region proposal network, which is part of the CNN architecture. So instead of using a static algorithm, the region proposal network can learn how to extract the regions. By canceling the classical region proposal algorithm, Faster R-CNN is significantly faster than Fast R-CNN.

One-Stage Detectors

In contrast to two-stage object detectors, which first create region proposals and then classify and fine-adjust the borders of a possible object, one-stage object detectors perform this in a single step. The aim is to improve performance over two-stage detectors because all computations are done in a single forward pass of the network. Notable members of these family are the YOLO networks [6, 38, 39, 7] and SSD [40].

Because YOLOv4 is the base architecture in this work, its structure and function principles are explained in more detail in a separate section (Sec. 2.4).

2.4 YOLOv4 Architecture

When Redmon et al. [6] came up with the idea of the YOLO object detection architecture, SOTA at the time were two-stage detector approaches. Although it scored lower on the VOC 2012 challenge [41] compared to the two-stage methods, it generalized better and was significantly faster. Hence, it became popular at that time.

Redmon worked on two consecutive YOLO iterations (YOLOv2[38] and YOLOv3[39]) before parting away from the computer vision research. Although Redmon himself was not directly involved in the following YOLOv4[7] architecture, it is often considered the “official” successor of the original YOLO branch. The reason is that the main contributor, Bochkovskiy, was already heavily involved in the original Darknet repository from Redmon.

Alongside the work of YOLOv4, YOLOv5[42] from a different group of authors came out short after YOLOv4. At the time of writing this work, there are already YOLOv7[43] and YOLOv8[42] available.

2.4.1 Network Structure

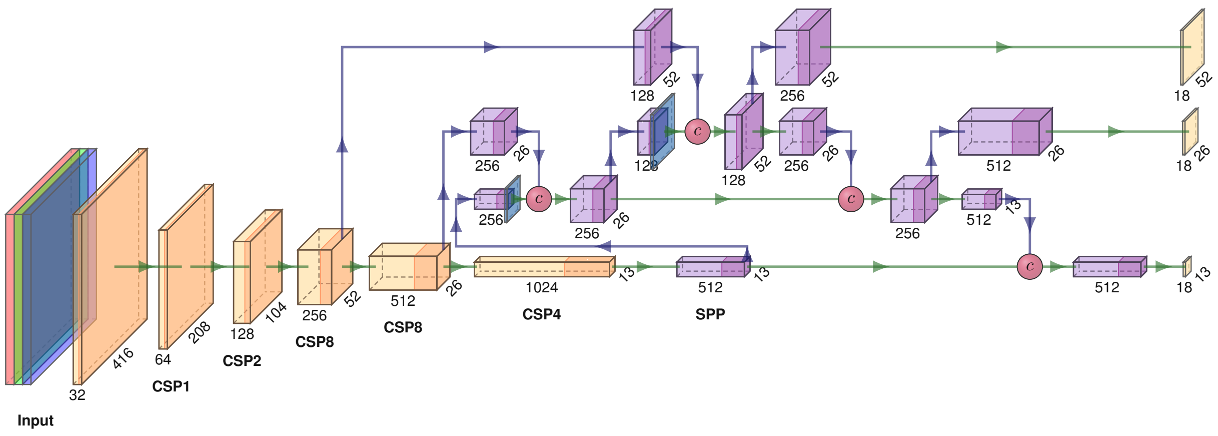


Figure 2.12: Abstracted overview of the YOLOv4 architecture. Plot created with [3].

The structure of the YOLOv4 architecture can be loosely sectioned into three main parts: The backbone, where the feature extraction happens; the neck, where features from different stages of the backbone are mixed; and the YOLO head, where the bounding box regression and the classification takes place.

The backbone of YOLOv4 is CSPDarknet53[44]. The backbone is further analyzed in Section 4.1. In the original network configuration from the YOLOv4 paper, the network input size is 512×512. It contains 29 convolutional layers with 3×3 kernels, a 725×725 receptive field, and 27.6×10^6 parameters. As mentioned in Section 2.2.3, the perceptive field describes how many input neurons affect a neuron in the output stage. A receptive field resolution larger than the actual input resolution can be interpreted that not only an output neuron can see the whole input, but also it is connected to all input neurons with multiple connections [7].

To enhance the perceptive field, YOLOv4 uses a Spatial Pyramid Pooling (SPP) block[45]. In this block, the last feature map of the backbone with the size 13×13 is processed by three maxpool layers with kernel size 5×5, 7×7, and 13×13, respectively. Same-padding is used, resulting in all outputs having the same spatial dimension as the input. These outputs are concatenated with the original feature map to be further processed.

A problem with feature extractors for object detection is that information about small objects is no longer available at the end. Features based on higher resolutions, extracted in earlier backbone levels, have since been processed in favor of a higher feature dimension. To tackle this, YOLOv4 uses a modified version of a Path Aggregation Network (PAN), which builds the network's net. The PAN takes the output from the SPP block and mixes it with two branches from earlier levels with higher resolutions. Once the resolution of the earliest branch is reached, the same steps are repeated in the other direction.

In the PAN of YOLOv4, the mixing is done by upsampling/downsampling the feature map and concatenating it with the branches, followed by a convolution layer to reduce the feature dimension. This method is slightly modified from the originally proposed method [46] by concatenating the features instead of summing them up.

The output of the neck consists of three branches with different spatial resolutions, going to the head (more precisely, three heads) of the network. YOLOv4 pursues a multi-scale approach with these separate heads: each head is designed to recognize objects of a specific size.

The YOLO heads themselves are introduced in YOLOv3[39]. Compared to two-stage detectors, YOLO does bounding box regression and classification in a single forward pass. The YOLO head works on an anchor-based approach. This means that the predicted parameters are not absolute bounding box coordinates but transformation parameters for predefined anchor boxes. Each YOLO grid cell outputs one bounding box prediction for each of the three anchor boxes. The transformations are as follows:

$$b_x = \sigma\left(t_x\right) + c_x \tag{2.22}$$

$$b_y = \sigma\left(t_y\right) + c_y \tag{2.23}$$

$$b_w = a_w \cdot e^{t_w} \tag{2.24}$$

$$b_h = a_h \cdot e^{t_h} \tag{2.25}$$

where t_x, t_y, t_w, t_h are the outputs of the NN, c_x and c_y is the top left corner of the current cell, a_w and a_h are the anchorbox's dimensions, and σ is the sigmoid function. An objectness score and the class probabilities accompany each prediction. In total, each cell outputs $(4 + 1 + N_{class}) \times 3$ predicted values.

2.5 Sensor Fusion in Neural Networks

Classical sensor fusion is a well-researched field. It is often done by fusing multiple sensor signals to form a new one. The fusion can be any mathematical operation, e.g., addition or weighted average. The input signals can either be from the same or different domains. A system can then process the resulting signal. An example is to fuse the signals of a temperature and a humidity sensor.

However, sensor fusion inside neural networks has the scope of doing the actual fusion operation inside the network. This results in two significant benefits: 1.) During training, the network can adjust the fusion to optimize it for the training data; 2) the network can also process information from the separate sensor signals.

Instead of doing a fusion inside the neural network, separate neural networks could also process the sensor signals separately. The results of the separate networks are evaluated and form the final results. Fusing results is often referred to as high-level fusion. There are two main shortcomings: One, forming the final results from the separate networks adds another design challenge where one must find a suitable algorithm. Second, processing the signals in parallel networks multiplies the needed computational resources. Especially for resource-constraint embedded platforms, this can be a leading factor for using a fusion approach.

A comprehensive survey [47] of deep multimodal object detection and semantic segmentation for autonomous driving stated that all neural network fusion approaches can be categorized based on three questions: 1) What to fuse?; 2) How to fuse?; and 3) Where to fuse?.

2.5.1 What to fuse?

In visual object detection with neural networks, the input signal typically contains images recorded in the visible light spectrum. When another signal domain shall accompany the images, often depth-information in the form of LiDAR, and sometimes Radar is used to allow 3-dimensional object detection [47]. Another often-used option is to add images recorded in another spectrum. The most often used spectra are infrared (LWIR, Mid-Wavelength Infrared (MWIR) and Short-Wavelength Infrared (SWIR)) [47]. LWIR is especially interesting for pedestrian detection applications because the body heat radiated from humans stand out in the images [8, 48].

2.5.2 Where to fuse?

Neural networks offer a wide variety of possible positions to fuse signals. Theoretically, fusion is possible in nearly all neural network stages (Backbone, Neck, Head). However, it seems from the studied work that most do fusion inside the backbone of object detection networks. Inside the backbone, there is also a differentiation possible regarding the position. However, the fusion is not always done in a single stage of the network but rather over multiple network layers. Hence sometimes, a more precise determination is not possible. Feng et al. [47] describe the position of the fusion as a tradeoff: Early-stage fusion can fully exploit the information of the nearly raw data and has the lowest computational costs but is unflexible in terms of modality changes (nearly the whole network need to be retrained) and susceptible in terms of spatial shifts between the modalities. Late-level fusion has the best flexibility (only partial retraining of the network is needed) but the highest computational cost, and it also cannot use (maybe useful) intermediate feature representation from earlier stages. Middle-level fusion is a compromise between the stated properties. The authors also point out that they found no conclusive evidence of a pattern, which position generally works best. They conclude that it highly depends on the application, data, and network architecture.

2.5.3 How to fuse?

The fusion can be implemented by a simple mathematical operation like addition or average or more sophisticated ones like additional layers in the neural network. According to Feng et al. [47], the most often used methods are addition/averaging; concatenation; ensemble; and mixture of experts. Concatenation of features is done by stacking the separate feature maps depth-wise before going to the next convolutional layer. Ensemble fusion is often used to union RoIs in object detectors [47]. Mixture of Experts is an approach where an additional network decides how to weigh the separate features.

2.6 Related Work

This section presents work that tries to solve the same questions with different methods or different questions with similar methods. To be more precise, this section focuses on papers that either investigate multispectral fusion schemes in a neural network, using YOLOv4[7] for fusion or study the impacts of fusion architectures on real hardware.

2.6.1 Multispectral Fusion Schemes

Liu et al. [49] propose a method for a multispectral fusion architecture based on Faster R-CNN[37]. They investigate several fusion schemes, including early, mid, and late. The KAIST multispectral dataset [8] was used for their experiments. The authors conclude that their mid-fusion approach has the best overall performance.

Zhang et al. [50] tackles the problem of spatial shift between the spectra with a new neural network architecture called *Align Region CNN*. For their approach, the authors introduce a *Region Feature Alignment* module to predict the shift between the spectra channels. They train their network on the KAIST multispectral dataset [8]. For this purpose, they created a new annotation dataset known in the literature as the KAIST *paired annotations*. In addition, the work contains a study on how the *RoI Jitter* strategy during training can improve the robustness against multispectral channel misalignments. Further, they introduced a fusion method to reweight the features of the separate spectra streams according to their confidence.

Zhang et al. [51] propose a new fusion method by *Cyclic Fuse- and Refine* blocks. The idea is to refine the features of a single spectrum with the fused features of multiple spectra several times. For the experiments, the authors use the KAIST multispectral dataset [8] and the FLIR ADAS dataset [52]. They state that the impact on inference time is only 0.4 ms with a Cyclic Fuse- and Refine block with a total of 3 loops, but they give no additional information on what hardware they take these measurements.

Zhang et al. [53] shows a multispectral fusion scheme using a multi-layer fusion structure with additional attention blocks. The authors use self-supervised trained channel-wise attention modules and spatial-wise attention modules trained by external supervision using saliency detection.

Wolpert et al. [54] propose a method for a single-stage anchor-free multispectral fusion architecture based on CSPNet[55]. The dataset used in this work is the KAIST multispectral dataset [8]. In addition, they study the impact between the *original annotations* [8], *sanitized annotations* [56] and *paired annotations* [50] of the dataset. They studied several fusion schemes, such as input-fusion, mid-fusion, and late-fusion. They identified *Random Masking*, *Random Erasing*, and *Noise augmentations* as feasible augmentation techniques for the multispectral data. This work delivers interesting results about the training methodology regarding annotations and augmentation of multispectral data and fusion impact on detected object sizes.

Farahnakian and Heikkonen [57] investigated three different fusion methods: pixel-level, feature-level, and decision-level. They conduct their experiments on a real marine multispectral dataset containing visible color and infrared images. The architectures are based on Faster R-CNN[37]. The mid-fusion (pixel-level) approach delivers the best results in their experiments.

2.6.2 YOLOv4 for Multispectral Fusion

Cao et al. [58] introduced a novel multispectral feature fusion operator called *multispectral channel feature fusion*. Their module can fuse the multispectral feature streams according to their illumination information. The authors study several fusion architectures based on YOLOv4[7], including early fusion, halfway fusion, late fusion, and direct fusion. They use the KAIST multispectral [8] and the Utokyo dataset [59] for evaluation and conclude that their halfway fusion achieves the best performance.

2.6.3 Hardware Impact

Nataprawira et al. [60] optimize the YOLOv3[39] architecture by adding a feature output stream to improve the detection of small pedestrians together with a multispectral input. The authors studied multiple network compression methods to optimize the fusion architectures for processing time. The proposed optimization techniques are all modifications to the neural network architecture itself and not target-hardware optimizations like tensorRT. These experiments regarding the processing time optimization are conducted on an NVIDIA RTX 2080Ti desktop GPU.

Roszyk et al. [61] adopt the YOLOv4[7] architecture with a multispectral input and optimize it for low-latency pedestrian detection. They use the KAIST dataset with the original, unmodified labels from [8]. They investigated sensor, early, middle, and late fusion schemes. In their work, the terms early-fusion and late-fusion refer to approaches before and after the neural network, therefor middle-fusion is the only fusion approach happening in the network. The authors include a comprehensive study on the impact of network latency and the ability to detect pedestrians at a certain distance. After comparing the results, they further optimized the approaches with tensorRT, achieving 35 FPS on the mid-fusion architecture and 410 FPS with the mid-fusion approach applied on a YOLOv4-Tiny. Although the authors optimize the networks with TensorRT, they only conduct measurements on the desktop GPU NVIDIA RTX 3080, not an embedded device.

Zhang et al. [62] introduces a new fusion operator called *Guided Attention Feature Fusion*, to dynamically fuse the multispectral feature streams. It does this by utilizing multiple attention modules in each spectra stream and also between the multispectral streams. The experiments are conducted on the KAIST multispectral dataset [8] and the FLIR ADAS dataset. Their work also includes an analysis of the runtime impact of their proposed fusion module on the NVIDIA GTX 1080Ti and the NVIDIA Jetson TX2. These results are achieved by running PyTorch on these devices, not including TensorRT.

Chapter 3

Methodology

This work aims to study the impact of fusion approaches on the YOLO4 architecture on the embedded NVIDIA Jetson hardware platform. The focus is optimizing these approaches with TensorRT and investigating their impact on object detection metrics and system performance. Before conducting experiments with fusion architectures, the developer must make several design decisions. This chapter outlines the selected framework and the chosen dataset regarding their properties. Furthermore, the training strategy and the training hardware are disclosed. As a vital part of the methodology, the optimization workflow from the trained NN model to the optimized TensorRT engine is described. The chapter ends with an explanation of how the measurements on the Jetson Xavier are taken.

3.1 Darknet Framework

For construction and training of the DNNs, the open-source Darknet framework [1] is chosen. It was initially developed by Joseph Redmon [63] to demonstrate the capabilities of YOLO. Since Redmon's departure from computer vision research after YOLOv3, the repository was forked by Alexey Bochkovskiy. This fork became the best-maintained version of Darknet.

The Darknet framework is mainly written in C and CUDA, enabling fast network inference. The usability with YOLO architectures is simple. The fact that the measurements in the papers (up to YOLOv4) were made using the darknet repository rules out framework-dependent variations when reconstructing results. Although designing models and training in Darknet can be considered easy, it has fallen behind PyTorch and TensorFlow regarding user-friendliness. It is not as easily extensible, and its installation and compilation process is much more complex than PyTorch or TensorFlow. Also, Darknet

[1] https://github.com/AlexeyAB/darknet

is primarily for object detection networks and doesn't offer the network variety of other deep learning frameworks.

```
[net]
batch=64
subdivisions=8
# Training
# width=512
# height=512
width=608
height=608
channels=3
momentum=0.949
decay=0.0005
angle=0
saturation=1.5
exposure=1.5
hue=.1

learning_rate=0.0013
burn_in=1000
max_batches=500500
policy=steps
steps=400000,450000
scales=.1,.1

#cutmix=1
mosaic=1

[convolutional]
batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=mish

# further network definition
```

Listing 3.1: Example darknet config file

The network definition and training config for Darknet networks are stored in a .cfg file by default. An example of the beginning of a typical config file[2] is shown in Listing 3.1. A complete network definition in Darknet consists of the network definition and weight files, where all the network parameters are stored. The information relevant for training and validation regarding the dataset(s) is stored in a separate (.data) file.

Due to its popularity, bridges between the Darknet model format and other frameworks exist in several repositories. Among them are bridges to ONNX and PyTorch. In addition, some tools can also load and infer networks directly from the Darknet model format, like OpenCV's DNN module [3].

3.2 KAIST Dataset

The KAIST dataset [8] consists of $95,328$ multispectral image pairs. The image pairs are recorded in the visible light spectrum and the LWIR spectrum.

A specialty of the image pairs in the dataset is the way they are recorded: The authors set up a recording rig with a beam splitter to enable the alignment of the optical image axes. Through a camera calibration process, they can pixel-align the spectra images. Often, fusion approaches expect pixel-aligned spectra images. If the dataset is not aligned, preprocessing is required to align them. Hence, aligned images allow neglecting this additional challenge in the experiments.

[2]https://github.com/AlexeyAB/darknet/blob/master/cfg/yolov4.cfg
[3]https://docs.opencv.org/4.x/da/d9d/tutorial_dnn_yolo.html

3.2.1 Image Properties

The image pairs are recorded with a framerate of 20 fps. The RGB images are recorded with a PointGrey Flea3 camera and an original resolution of 640×480 pixels. The LWIR images are recorded with a FLIR-A35 camera with an original resolution of 320×256 pixels. The infrared camera is sensible in the wavelength range $7.5 - 13\ \mu m$. The vertical field of view of the RGB camera with $103.6°$ is larger than the infrared cameras' with $39°$. Hence, after rectifying the images, the authors cut the outer parts of RGB images to match the field of view. The final processed images have a 640×512 pixels resolution. Example image pairs for day and night scenes are depicted in Figure 3.1.

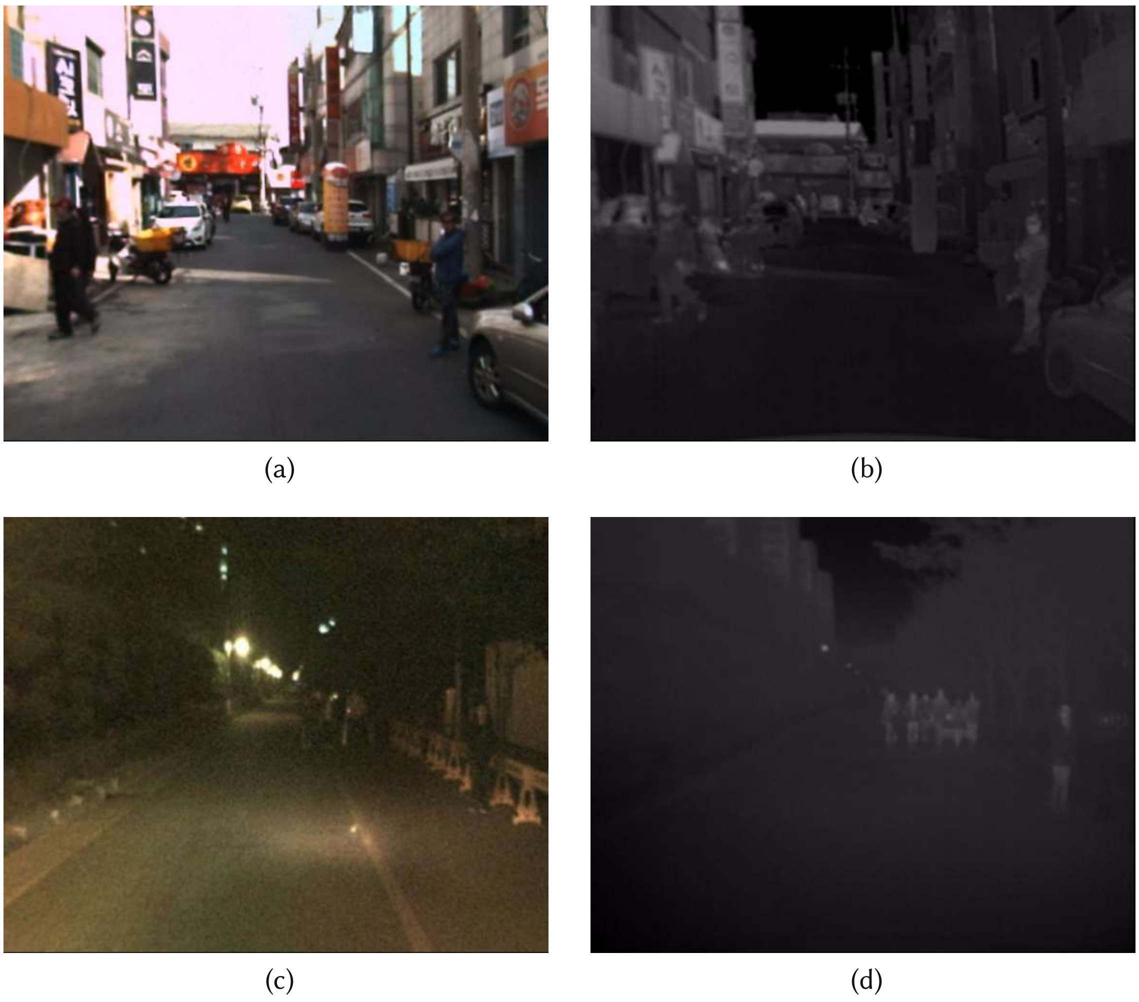


Figure 3.1: Example image pairs of the KAIST dataset: (a) Day scene RGB image; (b) Day scene LWIR image; (c) Night scene RGB image; (d) Night scene LWIR image.

3.2.2 Dataset Splits

There are a total of 95, 328 image pairs in the KAIST dataset. The images are organized into 12 scenes, each containing several sequences. The scenes contain day and night lighting conditions. The authors' criteria to split the dataset into train and test datasets were: First, both should contain a similar number of pedestrians. Second, the number of day and night images should be similar in both sets. They ended up with the following split: The train set contains the sets with indices 0-5, resulting in around 50 thousand images. The test set contains the sets with indices 6-11, resulting in around 45 thousand images.

3.2.3 Dataset Annotations

The intended use case of the KAIST dataset is for pedestrian detection. Hence, only visible pedestrians are annotated. The authors used similar labels as in the work of Dollár et al. [13]: Individual pedestrians have the label "person". Several pedestrians forming a crowd are labeled as "people". If it is unclear if the object is a real person or another object, it is labeled as "person?". A person riding a bicycle is labeled as a "cyclist". In addition, the annotations also have an extra field for how occluded the object is: no-occlusion, partial occlusion, and heavy occlusion. The original annotations are available in the VBB format and can be extracted to the bbGt format [64].

Alongside these original annotations from Hwang et al., several adopted label versions were published by other works. Li et al. [56] proposed the *sanitized* training annotations for the KAIST dataset. They are created by sampling every two frames from the original training set [65], excluding all occluded, truncated, or small (height less than 50 pixels) pedestrian instances, resulting in $7,601$ image pairs at least containing one valid person instance. These images are then manually relabeled. The authors modified the label categories in two ways: First, labels that are not spatially aligned between the RGB and the LWIR image are relabeled as "person?a". Second, the "cyclist" labels were relabeled to "person" because the human annotators of the paper found it difficult to distinguish them under certain conditions.

Zhang et al. [50] created the *paired* training annotations. They filtered the dataset to obtain $20,025$ valid frames and annotated $59,812$ pedestrian instances. The authors created separate labels to account for the spatial shifts between the spectra. The *improved* annotations from Liu et al. [66] exist for testing. They correct many errors from the original annotations. The authors created them by sampling the test annotations every 20 frames and manually relabeling them, resulting in $2,252$ image pairs.

3.3 Training Details

The training of the networks is entirely done with the built-in functionality of the Darknet framework. If not stated otherwise, the sanitized training annotations [56] are used for training, and the improved annotations [66] are used for training validation.

The training is conducted with the dataset split into train and validation sets. Not providing an additional test set can negatively affect the generalization performance. The reason is that a hyperparameter optimization will be biased onto the provided validation set - therefore, the performance on the validation set will be better than on never seen images. However, because hyperparameter optimization plays a subordinate role in this work, and the main target is to make relative comparisons inside the scope of this work, a split into two parts is sufficient.

To minimize the influence of training variations, two training runs are carried out for each network, and the best checkpoint was selected to continue with.

3.3.1 Annotation Filters

Although the human-sanitized version of the annotations has significantly better quality, the question arises if all of them should be used for the training. In other work, filtering the labels of the KAIST dataset before training is a common method. This intends to filter out objects that are difficult to learn and could potentially hurt the training progress overall. E.g., objects that are too small for the object detector or if the object detector is known to have bad performance on small object sizes. In favor of a realistic setting, all sizes and occlusions are used. In this work, only person labels are used.

3.3.2 Training Validation Policy

The default validation metric in the darknet framework is mAP as described in Section 2.1.1. It is calculated every few epochs and is used for detecting overfitting and determining the best training checkpoint (see Section 2.2.2).

Instead of the mAP, LAMR is selected as the target metric for the final evaluation in this work. The reasons for preferring it over mAP, besides that it seems to be the most often used metric in pedestrian benchmarks, are outlined in Section 2.1.1. Additionally, the LAMR should target the “reasonable” subset, which seems to be the primary comparison metric from reviewed work. Consequently, this means that the metric to determine the best checkpoint during training validation is also set to LAMR. This is necessary because due to the usage of the “reasonable” subset for the LAMR metric, while all labels are used for the darknet mAP calculation. Therefore it is unlikely that the checkpoint with the best LAMR value on the “reasonable” subset is also the best in terms of mAP.

To get the LAMR values during training, the validation code in Darknet is modified. In addition to the original mAP, the calculation for LAMR is added. The script for LAMR calculation is taken from

Kim et al. [67], as it was suggested[4] by the original author of the baseline paper [8]. A more detailed description of the methods used in the script can be found in Section 3.4.

3.3.3 Data Augmentation

Typically a neural network-based object detector is trained with so-called data augmentation. This means that the network is, in addition, fed with slight variations of the original images. These variations can be brightness, hue, saturation, noise, blur, cropping, etc. The idea is to create a larger dataset with more variety virtually. For example: If the network only sees good illuminated scenes during training, it probably has problems dealing with bad illuminated images afterward. If it sees some (artificially darkened) images during training, chances are that the network can handle these situations better.

For regular single spectra object detectors, data augmentation is almost always beneficial for robustness. However, augmentation for multispectral images opens a challenge. There is likely a relation between how one spectrum changes with the appearance of the other. A possible relationship could be modified if the data augmentation is randomly chosen on the spectra. The assumption is that augmentations need to be modeled instead of randomly chosen.

This work focuses on comparing different networks. If all are trained with the same (minimal) augmentation, all networks have the same advantage/disadvantage. Therefore all training runs in this work are conducted with only random network input size and mosaic augmentation.

3.3.4 Hyperparameter Optimization Strategy

Similar to the policy for data augmentation, only a little attention is paid to hyperparameter optimization. Instead, test runs are carried out to find suitable learning rates and time stamps in training to unfreeze the backbone.

It is to say that there is potential for better training results in a proper hyperparameter search. To get the best result for each fusion approach, the hyperparameter search would need to be carried out for every approach. Since one focus of this work is applicability, an intensive hyperparameter search is neglected in favor of little design effort.

3.3.5 Initial Weights

Using initial weights instead of randomly initializing is a common technique known as transfer learning. It allows the training to start from a point where the network can detect various typical objects.

[4]https://github.com/SoonminHwang/rgbt-ped-detection/issues/11

The networks used for initial weights are often trained on large, general datasets like ImageNet[68] classification and COCO[69] for object detection. The provided pre-trained weights from the Darknet repository are used to train the single-spectrum reference networks as follows: The used CSPDarknet53 backbone was first trained on the Imagenet classification dataset. This trained backbone was then placed in the YOLOv4 network, and the whole network was trained on the COCO object detection dataset.

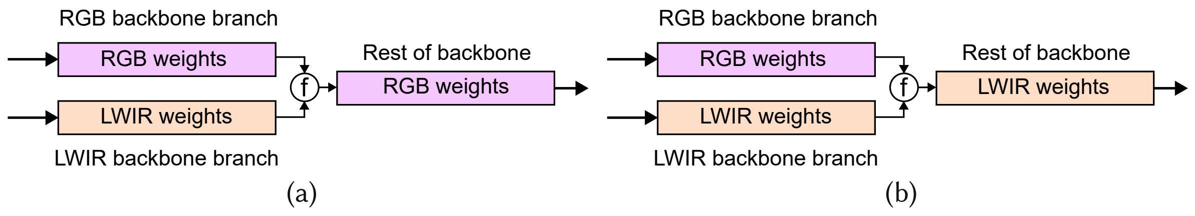


Figure 3.2: Initial weight scheme for fusion approaches. (a) Rest of the backbone is initialized with RGB pre-trained weights; (b) Rest of backbone is initialized with LWIR pre-trained weights

The weights from the reference single spectra networks are utilized as initial weights for the fusion networks. To be more precise, the weights from the RGB reference network are used for the RGB branch of the fusion backbone; the weights from the LWIR network are used for the LWIR branch. Doing so brings the advantage for the training that each branch has the capability to extract meaningful features right from the beginning. An open question is how the rest of the network weights should be initialized with RGB or the LWIR reference weights. Hence, both variants are conducted in the experiments section.

3.3.6 Training Hardware Setup

Training a neural network, in general, is a resource and time-consuming task. Depending on the used framework, dataset, network model, and computational resources, the training can last several days until it converges to a suitable solution. In this work, there are six fusion architectures, and each is carried out with several runs. Training them all one after another on a regular PC would be impractical regarding the needed time. A more convenient method is to utilize a computational cluster to train several networks in parallel. On a cluster, there are so-called nodes that can run multiple tasks independently. This work uses the Vienna Scientific Cluster (VSC)[5] in the expansion stages 3+ and 5. Both of them offer a variety of CPU and GPU nodes. For this work, all training runs are conducted on GPU nodes: NVIDIA A40 nodes of the VSC 3+ and the NVIDIA A100 nodes of the VSC 5.

The software and tooling stack to train a neural network with Darknet on the VSC looks like the

[5]https://vsc.ac.at/

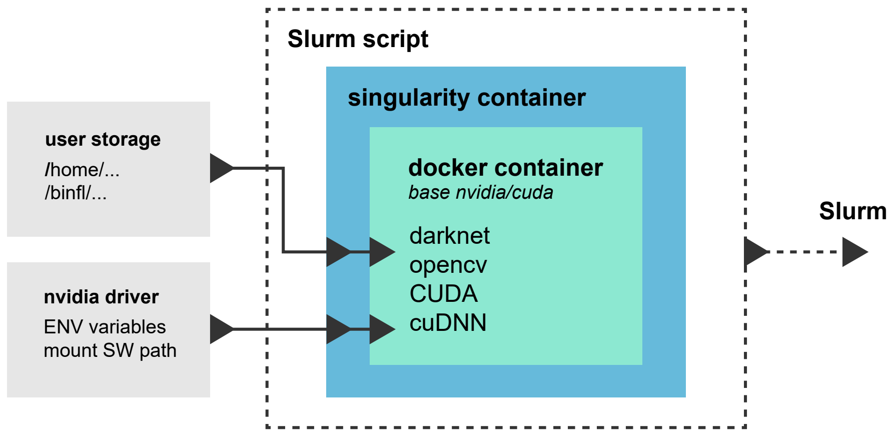


Figure 3.3: Illustration of the components needed for training on the VSC.

following: The Darknet framework and supporting software components, e.g., OpenCV, are compiled and installed in a Docker[6] image. The base for this image is the NVIDIA CUDA image, which contains the development libraries for CUDA and cuDNN. This Docker image is then transferred to the cluster and converted there into a Singularity[7] image file. On the VSC, Singularity is used instead of Docker because of the shortcomings of Docker when used on a multi-user system. The beforehand converted Singularity image file can then be used analog to a Docker image by the user.

Slurm is used as a job scheduler and resource allocation manager across the cluster. A helper script needs to be called to start a training run. The script first creates the according file structure, including training support files. After that, it queues the slurm job script into the cluster-wide job queue by requesting the resources. When the requested resources are free and allocated to the job, the script starts from the allocated computational node. The script starts a Singularity container with the mentioned image and calls the appropriate Darknet training commands inside the container. An illustration of how these software components on the VSC work together is depicted in Figure 3.3.

3.4 Evaluation Details

The LAMR metric is used to evaluate the object detection performance. The evaluation in this work is done on the "reasonable" and the "all" subset of the KAIST dataset. In both cases, only specific labels should be taken into calculation, e.g., only "person" labels. The question arises what happens to the evaluation when the other labels like "cyclist" are ignored? If a label to ignore is left in the ground truth and is not matched by a prediction, it would be a false-negative and degrade the LAMR metric. If it gets matched, it is a true-positive and would make the LAMR appear better than it is. On the other hand,

[6]https://www.docker.com/
[7]https://sylabs.io/docs/

if the label to ignore is removed from the ground truth and gets a matching prediction, it results in a false positive result, degrading the evaluation metric.

The problem of dealing with labels, which one is not interested in, is a well-known problem described in Dollár et al. [13]. The authors also describe rules for dealing with these “ignore regions” in that work. These also include how to deal with “people” labels, where the individual persons are no longer distinguishable. This ruleset was also used by the authors of the KAIST dataset [8]. A much more recent evaluation script was published in Kim et al. [67]. Because one of the KAIST dataset authors is also a co-author in that work, the mentioned evaluation script is also used in this work.

The evaluation script uses the pycocotools Python library as the backend. Hence it prefers to get detection data in the COCO format [69]. A function is implemented to export the detections during the validation to use the script independently of the Darknet framework.

3.5 NVIDIA Jetson Platform

The Jetson platform is a series of embedded computation boards by NVIDIA. It is dedicated to efficient embedded machine-learning applications. The boards are driven by NVIDIA’s Tegra System on Chip (SoC). The Jetson boards themself are in the form of a System on Module (SoM), meaning the integration in a custom carrier board design can be done via a socket. The Jetsons are also available as a development kit, including a carrier board with several IO interfaces, additional storage extensions, cooling, etc. The development kit enables rapid prototyping due to plug-and-play availability. The first Jetson generation, named TK1, was released in 2014. Since then, many subsequent generations have been published, including TX1, TX2, Nano, Xavier, and Orin.

The core of the Jetson platform is the TEGRA SoC. On a single chip, it combines a powerful GPU and an ARM CPU with other components like hardware video encoders/decoders, deep learning accelerators, etc. The Tegra SoCs, besides their usage in the Jetson platform, are also used for mobile devices and gaming handhelds[8]. The operating system is a modified Ubuntu version called Linux for Tegra (L4T). The software development tools needed for machine learning applications are bundled in NVIDIAs Jetpack SDK. It contains, among other software components, CUDA, cuDNN, the NVIDIA Docker runtime, VPI, and TensorRT.

[8]https://www.nvidia.com/de-de/drivers/tegra-3/

3.5.1 NVIDIA Jetson AGX Xavier

This work selects the Jetson AGX Xavier as a representative embedded hardware platform. The available model for the experiments is the version with 16 GB of memory (Specs in Fig. 3.1), which eventually got replaced by a 32 GB version and is no longer available for purchase. The variants available fur purchase at the time of writing this work are Jetson AGX Xavier (32GB), Jetson AGX Xavier (64GB), and Jetson AGX Xavier Industrial (32GB). Jetpack 4.6.2 (L4T 32.7.2) was installed on the system for all

Table 3.1: NVIDIA Jetson AGX Xavier (16 GB) Specification

	Jetson AGX Xavier (16 GB)
GPU	NVIDIA Volta with 512 CUDA Cores and 64 Tensor Cores 22 TOPS (INT8)
DL Accelerator	(2x) NVDLA 10 TOPS (INT8)
CPU	8-Core ARM v8.2 64-Bit CPU, 8 MB L2 + 4 MB L3
Memory	16 GB 256-bit LPDDR4x | 137 GB/s
Storage	32 GB eMMC 5.1
Power Targets	10 W / 15 W / 30 W

experiments. This Jetpack version includes CUDA 10.2 and TensorRT 8.2.

3.5.2 Power Consumption Measurement

An important factor is how a NN can affect the system's power consumption. Hence, all experiments are conducted while measuring the power draw of the system. On the Jetson AGX Xavier, there are a total of six power rails: VDD_GPU, VDD_CPU, VDD_SOC, VDD_CV, VDD_DDRQ, and VDD_SYS5V. The complete power for the Xavier SoM is provided via two power supplies, coming from the carrier board: SYS_VIN_HV is the main high-voltage input ($9V$ - $20V$), and SYS_VIN_MV is the medium voltage input ($5V$). The SoM internal power rails are created from these two voltage supplies: VDD_GPU, VDD_CPU, VDD_SOC, VDD_CV, VDD_DDRQ are generated from SYS_VIN_HV, while VDD_SYS5V is generated from SYS_VIN_MV.

The Jetson SoM provides a total of two power voltage/current monitor ICs in the form of INA3221[9], each providing 3 channels for measuring voltage and current. The power rails VDD_GPU, VDD_CPU, and VDD_SOC are connected to the first power monitor, while VDD_CV, VDD_DDRQ, and VDD_SYS5V are connected to the second monitor. The measured values of voltage, current, and power can be easily accessed by reading their corresponding *sysfs* nodes. The values for voltage and current are calculated from the last 512 samples, and the power value is calculated from them.

[9]https://www.ti.com/product/INA3221

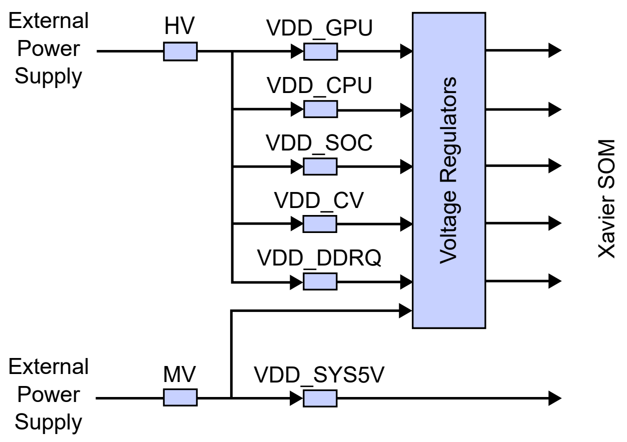


Figure 3.4: Power Rails on the Jetson AGX Xavier

A tool, already part of the L4T, is *tegrastats*, which delivers statistics of the whole system, such as CPU/GPU utilization, clock speeds, temperatures, and power consumption. The power measurement works with the previously mentioned power monitoring ICs. In addition, tegrastats also has the ability to calculate a running mean from the point in time the service gets started. This behavior is used by restarting the service before the measurements, forcing the tegrastats' internal average history to reset. After restarting the service, the network inference is called in a loop to create a steady power consumption. After 10 minutes, the tegrastats service is stopped, which disables the logging.

For the experiments in this work, the following methodology to measure the average power consumption is implemented in the form of a bash script:

1. Stop tegrastats service
2. Start tegrastats service in the background
3. Start continuous network inference
4. Stop tegrastats logging after 10 minutes

After the script has finished, the logged average power values can be extracted from the log for further evaluation. The experiments are conducted while not setting a specific system power target (MAXN nvpmodel). This ensures that no artificial limit is introduced, influencing the power and latency measurements.

3.5.3 Inference Measurement

To measure the inference time of the networks, precautions are taken to ensure that the conditions of the system are as identical as possible. The overall load of the whole system is a vital element in this. When running neural networks without outer constraints, some internal system components will likely act as bottlenecks. Hence, it is important that no other user process is running during the inference measurements.

Another crucial part is the Dynamic Voltage and Frequency Scaling (DVFS) governer. It scales the system components' clocks and voltages to match the current system load. When doing network inference, it is likely, that the system clocks are not stable during the measurement phase, which will likely influence the outcome. To tackle this, in this work's experiments, the DVFS is disabled before the measurements. This effectively sets all system clocks to their possible maximums. On a Jetson system, this is done via a script called *jetson-clocks*. In addition, to avoid bottlenecks caused due to power limitations, the power target of the whole system is set to unlimited. This is done on a Jetson system by setting the *nvpmodel* to mode 0 (MAXN mode). After setting up the system in the described fashion, the network inference itself runs in a loop of 1000 iterations after a warmup of 500 iterations. The mean of all measured times is taken for further evaluation.

Summary of the inference time measurement methodology:

1. Set power target to unlimited (MAXN, Mode 0)
2. Disable DVFS governor (jetson_clocks)
3. Warmup 500 network inferences
4. Measure times of 1000 inferences

3.6 TensorRT Optimization

TensorRT[10] builds the core of efficient NN model inference on the Jetson platform. It was developed to fully utilize the computational capability of the underlying hardware. To do this, TensorRT first builds an optimized model, the so-called engine. The engine can then be serialized and saved on the disk for later usage.

While building the Engine, TensorRT uses several optimization techniques, such as combining layers and operations, folding constants, eliminating unnecessary operations, and reordering operations for efficient computing on the GPU. In addition, it also can adjust the precision of the underlying computations to either 16-bit floating point or via a quantization to 8-bit integer type. Furthermore, TensorRT has a set of possible algorithms (called tactics) to optimize a specific network layer type. It uses benchmarking to find the best algorithm, then uses it for optimal execution scheduling, minimizing the cost of kernel executions and format transforms. These benchmarks depend on the system the engine is built on and the specific TensorRT version. Therefore, the engine has to be rebuilt for every system with different hardware or TensorRT version.

[10]https://developer.nvidia.com/tensorrt

By default, TensorRT pushes the executions onto the GPU. However, if the Deep Learning Accelerator (DLA) supports the computations, TensorRT can be introduced to use the DLA instead of the GPU. If the DLA does not support all network layers, a GPU fallback for these specific layers can be configured. Regarding the user interface, TensorRT comes with C++ and Python APIs. In addition, an easy-to-use command line tool called trtexec is provided for easy experimenting. For deeper debugging, there is also a tool called Polygraphy[11] available, which provides advanced functionality over trtexec.

For deploying a custom model with TensorRT, at the time of writing this, there are four main options to convert a custom model to an engine:

TensorFlow - TensorRT Integration (TFT-TRT) The TensorFlow[12] integration of TensorRT creates subgraphs in a TensorFlow model that TensorRT can accelerate. The rest of the graph is still executed natively in TensorFlow. The TensorFlow model can still be executed as usual.

Torch - TensorRT Integration (Torch-TRT) The Pytorch compiler of TensorRT selects subgraphs of the PyTorch model that TensorRT can accelerate. The rest of the graph is natively processed in Torch. The PyTorch module can still be executed as usual.

Build engine from an ONNX model ONNX is the primary model import format. TensorRT uses a built-in ONNX model parser for that. The conversion from an ONNX model allows the creation of better-performing engines with less overhead. However, all layers must either be supported by TensorRT or provided via a custom layer plugin. This approach provides advanced control of the resulting TensorRT engine.

Manually design TensorRT network graph Manually designing a TensorRT network graph of your target model with the API brings the best performance because of the fine-grained design options. At the same time, it is the least flexible approach because you have to manually adjust the created network graph if model changes are necessary.

3.6.1 Darknet to TensorRT Workflow

The models in the Darknet framework are defined by a combination of a .weight and a .cfg network definition file (as described in Section 3.1). There is no direct path to convert a model represented in the darknet model format to a TensorRT engine. A commonly used workflow is first to convert the Darknet model to an ONNX model and then build the TensorRT engine from it (Subfigure 3.5a). However, the Darknet repository from Bochkovskiy et al. does not support exporting to ONNX, so another repository must be used for the export. A possible repository to export the model to ONNX is

[11]https://github.com/NVIDIA/TensorRT/tree/main/tools/Polygraphy
[12]https://www.tensorflow.org/

the YOLOv4 Pytorch implementation [13]. A downside of this method is the possibility that by using an ONNX parser, some layer functionality or layer configuration gets lost or is not supported due to parser inabilities. The other workflow uses a direct conversion from the Darknet model layers to the according

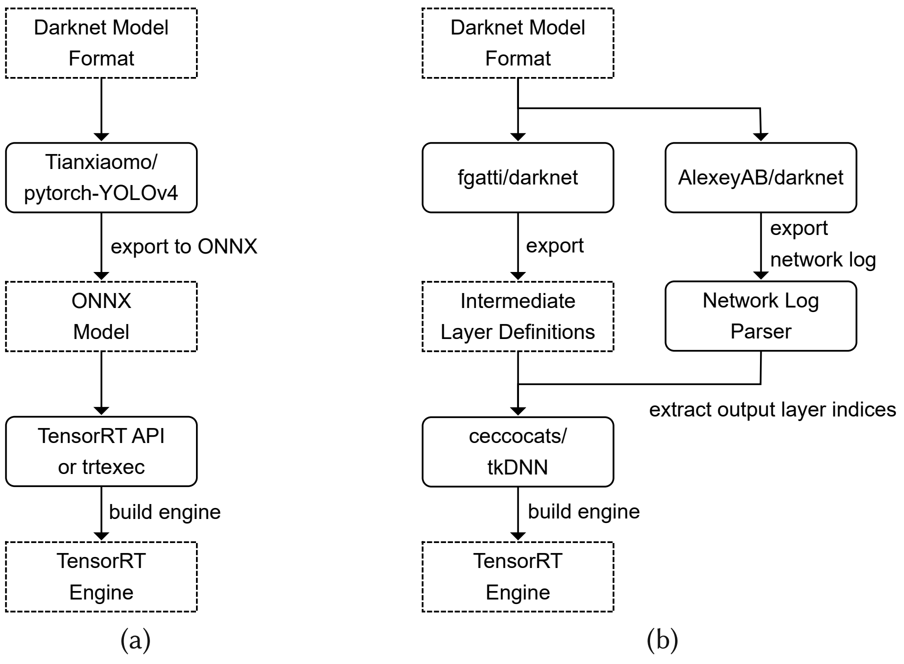


Figure 3.5: Flowchart of the conversion methods from Darknet to TensorRT models: (a) Conversion flow via the ONNX model format; (b) Conversion flow via the tkDNN library;

TensorRT primitives (Subfigure 3.5b). Instead of parsing the darknet model to get a resulting ONNX model, the Darknet layers are directly converted to the corresponding TensorRT layers. A library with such described capabilities is tkDNN[14]. It was proposed in the scope of the work of Verucchi et al. [70]. The authors describe tkDNN as "an open-source Deep Neural Network library built with cuDNN and tensorRT primitives, specifically thought to work on NVIDIA Jetson Boards, whose main goal is to exploit those boards as much as possible to obtain the best inference performance." The authors also state that the goal of tkDNN is not to be more efficient or faster than the engine building from an ONNX model but to have finer-grained control of the conversion process. The concerns about parser inabilities described for the ONNX parsers are also valid for tkDNN.

To utilize tkDNN to convert the YOLOv4 fusion networks, the first step is to export the Darknet model to an intermediate format consisting of layer definition and weight files. For this reason, one of the authors of [70] provided a Darknet fork [15] with this particular functionality. The second step is building the TensorRT engine with tkDNN. For this step, the output layers must be specified explicitly. In this work's

[13]https://github.com/Tianxiaomo/pytorch-YOLOv4
[14]https://github.com/ceccocats/tkDNN
[15]https://git.hipert.unimore.it/fgatti/darknet

networks, the output layers' indices differ for all fusion architectures because the number of layers also differs. A parser is introduced to allow this step's automation for varying network architectures. This parser uses the log output from Darknet to determine and set the correct YOLO output layers.

Short experiments with both workflows showed that the ONNX workflow (Subfig. 3.5a) has some shortcomings for some layer configurations used in the fusion architectures. Therefore, all TensorRT engines built in the scope of this work are using the workflow with the tkDNN library, depicted in Subfigure 3.5b.

Chapter 4

Fusion Architecture Search in YOLOv4

The selection and design of the fusion architecture play a critical role in how it will behave. This work aims to compare different fusion architectures against each other in terms of their properties. Hence, the question arises of how to select specific fusion architectures for the experiments. As mentioned in 2.5, the design space is too broad to compare all design aspects properly due to the lack of generally applicable design rules. The design space has to be restricted to obtain a manageable number of networks for comparison. This work realizes this by making some parameters invariant while allowing others to vary.

First, this section gives a more detailed analysis of the YOLOv4 backbone, and the main ideas behind the methods used are briefly explained. Secondly, the selected fusion networks are presented, including insights into why specific design decisions are made. Finally, it is explained how these architectures are implemented into the Darknet framework.

4.1 Analysis of YOLOv4 backbone

4.1.1 Structure

The backbone of YOLOv4 is CSPDarknet53. It is based on the Darknet53 network architecture, introduced in YOLO3 [39]. A key characteristic added to the CSP backbone is the name-giving cross-stage partial [44] connection: Instead of feeding all data through the dense layers, it is split into two parts at the beginning of each block. One part is guided through the dense layers, while the other is forwarded to the end of the stage, where it is recombined. Applying these CSP blocks to known architectures can reduce the computations needed while keeping or even enhancing accuracy [44]. In the CSPDarknet53 backbone, there is a total of five CSP blocks (depicted in Fig. 4.1). Their indices refer to the number of repeated Res blocks inside of it. The figure also shows a more detailed illustration of the CSP connection. Notice here that the convolutional layers include a batch norm and an activation layer if not stated otherwise. Generally, the backbone starts with a single convolutional layer right after the input layer,

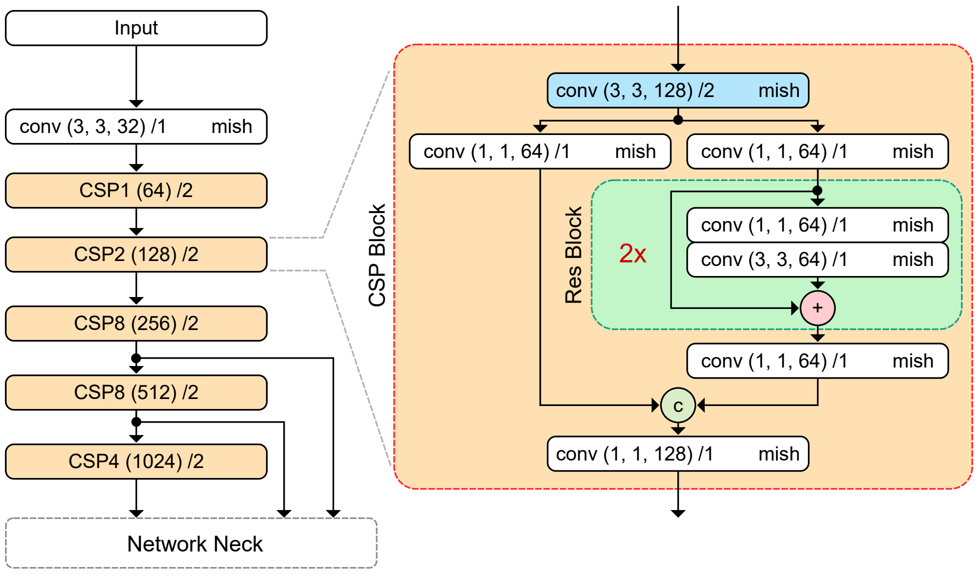


Figure 4.1: CSPDarknet53 backbone: Abstracted backbone structure with a detailed view of a CSP block, including residual blocks.

before the first CSP block. The first layer in a CSP block is a 3×3 convolutional layer which halves the spatial resolution and doubles the number of feature channels. This is the only layer with a spatial downsampling effect in the CSP block. After that, the feature map is split into two separate parts. One of them goes through the inner layers of the block, where the Res-blocks are located. The other is guided via a 1×1 convolutional layer directly to the end of the CSP block. The two formerly separated feature maps are recombined with a feature concatenation operation along the channel dimension. This newly created stack goes through a 1×1 convolutional layer.

In the CSPDarknet53, each of the CSP blocks contains one or multiple residual blocks (Sec. 2.2.2), which are also depicted in Figure 4.1. The first layer inside a Res Block is a 1×1 convolutional layer, followed by a 3×3 one. The input signal of the corresponding Res block is bypassed to the end of the block. But a feature addition operation is used in contrast to the CSP block. According to the specific CSP block, the Res block inside is repeated up to 8 times.

4.1.2 Feature representation

Understanding how the intermediate features are represented in the backbone is of significant interest for backbone fusion operations. Like in every other backbone, the goal of the backbone is to extract meaningful features from the network's input (Sec. 2.3.1). Typically, the spatial dimension is reduced to balance the computations needed for inference while the feature dimension is increased. A commonly used tactic is to double the feature channels while halving spatial dimension. The CSPDarknet53 backbone has five spatial dimension halvings - one inside each CSP block. This means that at the end of the last CSP block, right before the SPP block, the spatial resolution has $\frac{1}{2}^5 = \frac{1}{32}$ of the input resolution.

The extracted features in these intermediate layers are correlated to the spatial position of the related objects in the previous layer. Which node outputs can be influenced by the object is described through the concept of the perceptive field: The nodes from the previous feature map that lie in the perceptive field can change the corresponding node of the current layer. In simple terms: the extracted features observed in a layer node belong to an object that appears in the corresponding perceptive field in the image input.

4.1.3 Feature Fusion Design Aspects

In the context of feature fusion, the extracted features for an object in the input image should be assigned to the same positions in all spectra. This is because when fusing feature maps, one wants to exploit the information on how extracted features for a specific region are related. If the features are not aligned, this can degrade the relationship - either during training in the form of learning incorrect relations or during inference in interpreting it wrong. Therefore, the extracted features from the different spectra must be spatially aligned to a certain amount before fusing them. The amount depends on the fusion layer: the earlier the fusion appears, the smaller the perceptive field of the feature map nodes is. Hence, fusing on earlier positions requires higher spatial alignment than in the later layers. Another design aspect is the additional resource demand and latency, which a feature fusion approach has. For every additional layer added, the properties of the whole network change. To estimate the effect a fusion architecture will have on the system, it would be necessary to understand the resource demand and the latencies of the separate backbone layers in the non-fusion case. To study this, profiling tools like TREX[1] are available. TREX is an engine profiler for TensorRT models capable of generating comprehensive statistics of the engine's runtime properties.

But even with that information available, estimating a parallel network branch's influence is tough. The estimation after the TenorRT optimization is even more challenging because the network layers will likely get scheduled differently to balance the changed network. A precise estimation does not seem practically applicable during the design phase.

[1]https://github.com/NVIDIA/TensorRT/tree/main/tools/experimental/trt-engine-explorer

4.2 Fusion Architectures

4.2.1 Selection Aspects

The primary goal of this thesis is to analyze the influence of multispectral fusion architectures in neural networks, focusing on their comparative performance against non-fusion architecture networks and the impact of these architectures on resource utilization on an embedded device. The available literature on multispectral feature fusion in neural networks presents numerous approaches that, as discussed in Section 2.5, can be characterized by the three fundamental questions: what types of features to fuse, where to fuse, and how to fuse. While selecting the KAIST dataset (RGB + LWIR) in Section 3.2 implicitly answers the first question, the other two remain unresolved. The forthcoming paragraphs describe the design choices to address these questions.

Fusion Position

From researched literature, it seems that for fusing multispectral features, a common way is to start with separate branches for the spectra and fuse them at a designated point inside the network. A multilevel fusion approach is often utilized, whereby the separate backbone branches are merged at various layers throughout the network. This approach enables the fusion of information from different spectra at multiple feature extraction stages.

For this work, the selected architecture to investigate is the first type. The primary reasons for choosing this approach, while prioritizing practical applicability, are its ease of implementation. While multilevel fusion is relatively straightforward, the first type's simplicity with only one fusion point makes it preferable in design and during training. Additionally, this approach eliminates the need for extra design parameters related to the number and location of fusion points, further streamlining the implementation process.

The layer after which the feature fusion occurs is now the primary adjustment variable. The CSPDarknet53 backbone comprises 5 CSP blocks, so the positions between them are well-suited as potential fusion points. This results in a total of six fusion positions throughout the whole backbone. To denote the fusion positions within the backbone, arbitrary names are assigned to six points: the first two fusion points are referred to as Early Fusion I and Early Fusion II, the following two points as Mid Fusion I and Mid Fusion II, and the last two points as Late Fusion I and Late Fusion II.

Fusion Operator

There exists a wide range of possibilities for the fusion operator itself. From studied papers in this work and backed up through [47], the most often used operators to fuse features are concatenation, addition, and Mixture of Experts. The concatenation operation, sometimes called Network in Network (NIN), can be considered a weighted local average of the individual spectra feature maps. Generally, the separate feature maps of the spectra are concatenated, or stacked, along the channel dimension. To maintain compatibility with the original backbone layers, a convolutional network layer is introduced following the concatenation, which reduces the channel dimension back to its original size. The addition is a reasonably simple operator that sums up the corresponding elements in the feature maps without further processing. Mixture of experts is a technique that involves assigning a specialized network part (expert) to each input, expecting each expert to excel at handling specific inputs. A gating network, which is a trained neural network, takes inputs from specific network parts, such as the inputs images themselves or a layer preceding the fusion point, and calculates the weighting factors that should be applied to the outputs of each expert.

In this work, the concatenation fusion operator has been chosen primarily due to its ease of implementation and minimal computational overhead. Additionally, the operator does not require any design decisions, further simplifying the fusion process.

4.2.2 Selected Architectures Overview

Concerning the initial design decision, there is a total of six fusion architectures for conducting experiments. They all implement the same fusion operator but in six different positions. The following is a short overview describing their basic properties.

Early Fusion I

The Early Fusion I architecture (Fig. 4.2a) has the fusion point after the first convolutional layer, which brings up the feature map depth to 16 channels, and right before the first CSP block (CSP1). In the fusion point, the feature maps have the original spatial resolution of the input at a depth of 32.

Early Fusion II

In the Early Fusion II architecture (Fig. 4.2b), the fusion takes place between the first (CSP1) and the second (CSP2) CSP block. In the fusion point, the feature maps have a spatial resolution of $\frac{1}{2}$ of the input resolution at a depth of 64.

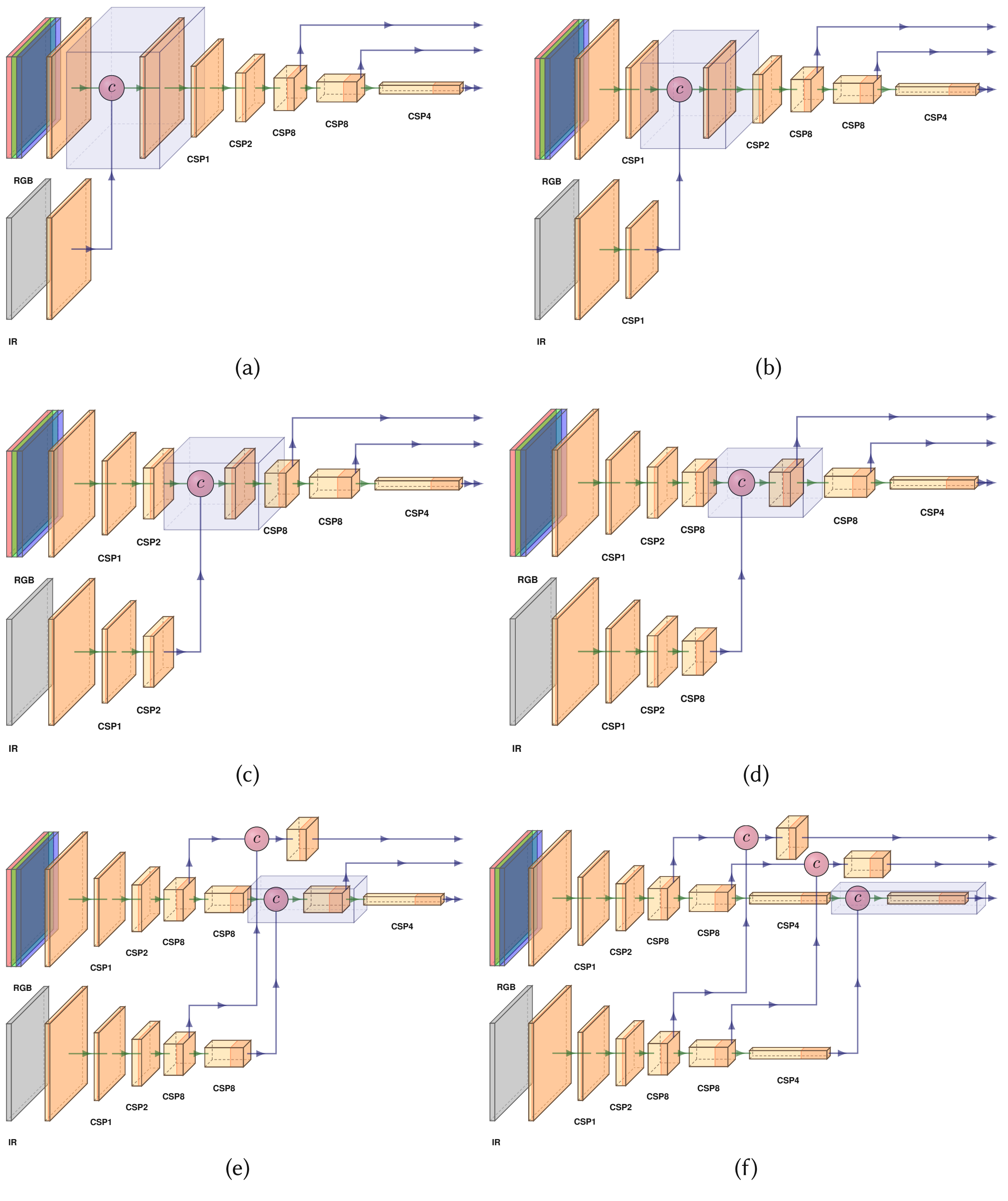


Figure 4.2: YOLO4: Selected backbone fusion architectures: (a) Early Fusion I; (b) Early Fusion II; (c) Middle Fusion I; (d) Middle Fusion II; (e) Late Fusion I; (f) Late Fusion II; Plots created with [3].

Middle Fusion I

The Mid Fusion I architecture's (Fig. 4.2c) fusion sits between the second (CSP2) and the third (CSP8) CSP block. The spatial resolution of the feature maps is $\frac{1}{4}$ of the original resolution at a channel depth of 128.

Middle Fusion II

The Mid Fusion II architecture's (Fig. 4.2d) fusion sits between the third (CSP8) and the fourth (CSP8) CSP block in the backbone. The spatial resolution of the feature maps is $\frac{1}{8}$ of the original resolution at a channel depth of 256.

Late Fusion I

The fusion point of the Late Fusion I architecture (Fig. 4.2e) is between the fourth (CSP8) and the fifth (CSP4) CSP block. The feature maps have a resolution of $\frac{1}{16}$ of the original resolution and a channel depth of 512. In contrast to the other architectures, the first branch going to the neck of the network needs to be fused.

Late Fusion II

The Late Fusion II architecture (Fig. 4.2f) has the fusion point after the fifth (CSP4) CSP block. The spatial resolution of the feature maps is $\frac{1}{32}$ of the original resolution at a channel depth of 1024. Additional to the fused branch running to the neck, the second branch must be fused.

4.3 Implementation in Darknet

The darknet framework does not provide all the necessary functionality required to support the presented fusion architectures without modification. Several changes need to be made to enable support, including implementing 4-channel input image support, loading different pre-trained weights into a single network, and in-training validation with the LAMR metric instead of the mAP. However, it should be noted that some components, such as the fusion operator, can be implemented without additional modification.

4.3.1 Multispectral Network Input

As Darknet does not (at the time of conducting the experiments) support multi-input networks, a challenge arises when attempting to feed multispectral images into the network. This thesis addresses this issue by utilizing 4-channel input images in PNG file format, with three channels representing RGB data and the fourth channel representing LWIR data. This allows for providing a single image containing all spectra (4 channels), which can then be separated into individual channels within the network.

Another consideration is the decision to train the LWIR reference network using three identical channels instead of a single channel. This is explained in more detail in 5.1. As a result, in the multispectral network, the single LWIR channel must be converted back into three channels to enable the use of pre-trained weights from the reference network. To achieve this, the route blocks of the Darknet framework are utilized. The four channels are initially split into individual channels and subsequently reconstructed into 3-channel RGB and 3-channel LWIR. In addition, an upsampling block with a stride of 1 (essentially a dummy block) is placed right after the network input because it is impossible to route directly from the input layer in Darknet. The resulting input structure is identical in all fusion networks and is depicted in Figure 4.3.

4.3.2 Miss Rate Training Validation

Darknet includes a built-in mAP calculation function for in-training validation purposes. However, since the target metric for this thesis' experiments is the LAMR, it is advisable to use the same metric for determining the optimal model checkpoint, as discussed in Section 3.3.2. There are two options for achieving this: implementing an LAMR evaluation function using the same logic as the Python evaluation script used in this project. The other option is directly implementing the Python script using the Python/C API [2]. The code is modified to save the detected objects during the validation phase in the COCO format, which can then be used directly for calculating the LAMR via the script.

The calculated mAP during the training's validation phase is displayed in the training graph, which can be observed to keep track of the training. In addition, to evaluate the performance of the models based on the LAMR metric, it is also included in the training graph.

4.3.3 Fusion Operation Implementation

The fusion implementation involves adding a route layer that concatenates the features along the channel dimension. This is followed by a convolutional layer with a kernel size of 1×1 and half the stacked channels as depth. As an example, the added fusion part for the Early Fusion I architecture is highlighted with a red dotted box in Figure 4.3b.

4.3.4 Pretrained Weights Loading

As mentioned, the fusion architectures are trained using pre-trained weights from both reference networks rather than randomly initialized weights. The various methods are outlined in Section 3.3.5.

To enable the loading of weights from multiple pre-trained networks in a flexible way, a new function is implemented that parses a weight configuration file. This file allows the specification of which layers of the network should be loaded with which weights, as well as the convolutional layer used for fusion,

[2]https://docs.python.org/3/c-api/

which can be excluded. This is necessary because darknet weight files are simple serialized versions of network parameters, and loading pre-trained weights into another network architecture would likely cause errors. It is important to note that the weights must always be loaded from the beginning of the separate weight files, so if only the last layers are needed, the first layers must also be written and then overwritten. To maintain as much compatibility with the original Darknet codebase as possible, this limitation has been imposed. This new weight configuration file provides a way to easily customize the loading of pre-trained weights for different fusion architectures.

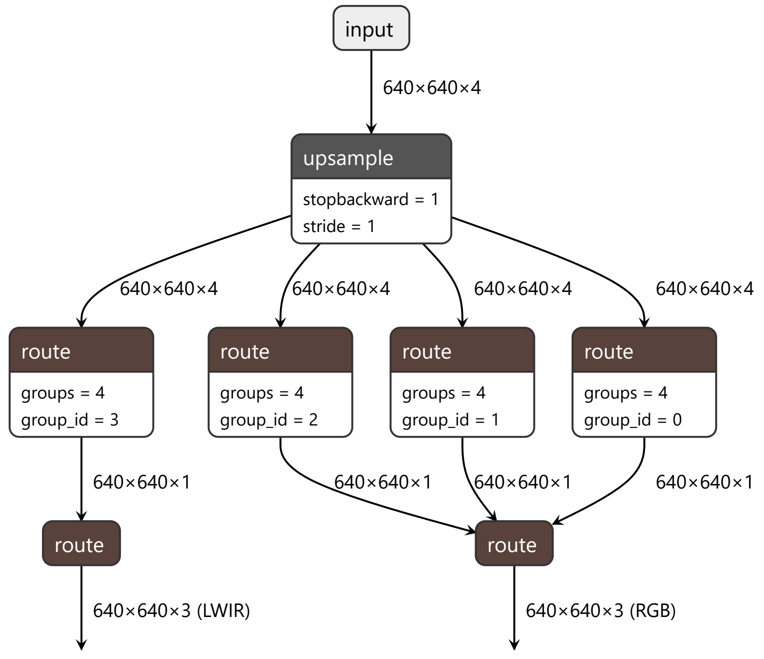
```
./yolov4_ref_rgb.weights    0    6      -1
./yolov4_ref_ir.weights     6    147    10
```

Listing 4.1: Weight configuration file for Early Fusion I with LWIR majority weigths

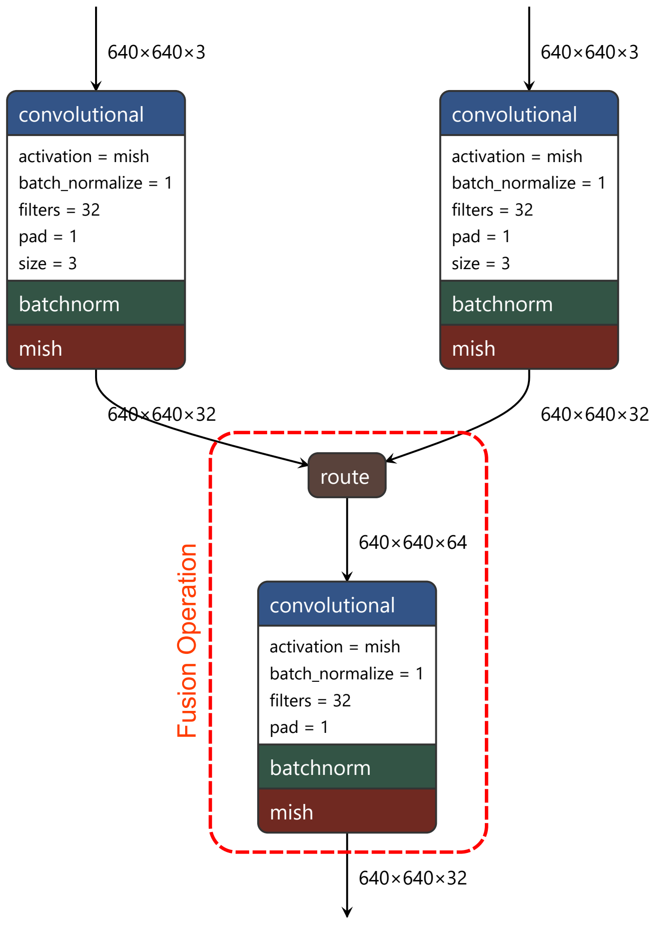
```
./yolov4_ref_rgb.weights    0    6      -1
./yolov4_ref_rgb.weights    6    147    10
./yolov4_ref_ir.weights     6    10     10
```

Listing 4.2: Weight configuration file for Early Fusion I with RGB majority weigths

In Listing 4.2, the weight configuration file for the Early Fusion I architecture with most RGB weights is depicted. The backbone's RGB branch (layer indices 0 - 5, only index 5 is a convolutional layer) is loaded with the first convolutional layer of the pre-trained RGB reference network. Next, the IR branch (layer indices 6 - 9) and the remaining part (layer indices 10 - 146) of the backbone, which follows the IR backbone branch in terms of layer indices, are also loaded with RGB weights. Note that index 10 is marked to be ignored because this convolutional layer is included in the fusion operation and should be initialized randomly. Finally, the reference IR network's first layer is loaded into the IR branch's position (6 - 9, where only layer 8 is a convolutional layer) in the fusion architecture.



(a)



(b)

Figure 4.3: Darknet: (a) Multispectral network input structure; (b) Continuation for the Early Fusion I network, including fusion operator; Plots exported from Netron [71].

Chapter 5

Experiments and Results

Following the explanation of feature fusion principles and the subsequent introduction of specific fusion architectures based on YOLOv4 in the middle section, numerous experiments are undertaken to examine the characteristics of these feature fusion architectures. A key question is how they perform compared to the non-fusion single-spectrum reference networks. This section presents the experiments conducted along with their corresponding results. The experiments are introduced concisely, highlighting their methodology and their main findings. The findings are further discussed within the corresponding experiment sections, providing a partial analysis. The section's end is a comprehensive summary that considers all experiments from the point of view of applicability.

First, the reference networks are introduced as a baseline. The reference networks are trained on single spectra images from the chosen dataset. Based on the weights of these reference networks, the proposed fusion architectures are trained. Afterward, several experiments are undertaken to study the impact of the multispectral feature fusion on latency, power consumption, and object detection performance. All experiments are conducted on a Jetson AGX Xavier embedded device.

5.1 Training of Reference Networks

The first important step is the introduction of baseline models. The baseline models in this thesis represent typical object detection models trained on a single spectrum. In a real-world use case, this would usually be the visible light spectrum or, in rarer cases, an infrared band. For this work, two reference models are introduced: one trained on the visible light images and one on the LWIR images from the KAIST dataset. They are used to compare the investigated feature fusion architectures for the experiments against the single-spectrum networks.

```
[net]
batch=64
subdivisions=16
width=640
height=640
channels=3
momentum=0.949
decay=0.0005
angle=0
saturation = 1.0
exposure = 1.0
hue=0

learning_rate=0.00001
burn_in=1000
max_batches = 38000
policy=steps
steps=9500,28500
scales=.5,.5

mosaic=1

# Further network definition

# last YOLO layer
[yolo]
random=1
```

Listing 5.1: Darknet config training file for the reference networks

Network Size

The network architectures for the reference correspond to the original YOLOv4 architecture configuration. The resolution is set to 640×640 pixels for both reference networks. This decision is made because the images from the dataset have a resolution of 640 × 512. Notice here that choosing a non-square network size, in general, is also possible, and the Darknet repository used for training in this work from [7] also supports non-square network sizes. However, it was not clear if all other repositories needed in the conversion workflow also support square sizes, thus leading to the design decision of using a square resolution - even though this adds unnecessary operations to the network. Or in other words: the same result could be achieved with a smaller network resolution. However, since the primary focus is on comparing the networks, and they all have the same resolution, the approach is justifiable.

Augmentation

Because of the design decision to neglect augmentation techniques for the fusion architectures (see Sec. 3.3.3), the reference networks are also trained with minimal augmentation techniques. Otherwise, there would be a situation where the reference networks perform better due to augmentation while the fusion networks do not have this advantage. This would distort the comparison. Therefore, the same augmentation techniques are applied to all networks to ensure a fair evaluation. To summarize, augmentations targeting optical image characteristics, e.g., saturation, exposure, and hue, are disabled for all training runs. The random network size variation is activated, allowing the network to change its input size to ±40%, making it more robust against varying object sizes. Furthermore, mosaic augmentation[7] is enabled. Mosaic augmentation is a technique where training input images are composed of four individual images, and the size of the four patches varies. This technique is used to learn different image contents, object sizes, and positions. All augmentation settings for the reference networks are depicted in Listing 5.1.

Input Channel Interpretation

While training the RGB reference network with three input channels (R, G, and B) is evident, the question arises of how the infrared input should be interpreted. The information in LWIR images is only stored in one channel, representing each pixel's intensity. This would make training the LWIR network with a single channel the obvious choice. However, the infrared reference network in this work is also trained with three input channels, consisting of the intensity channel plus two duplicated channels. The main advantage is that the pre-trained weights from the Darknet repository [7], which are created for 3-channel RGB input images, can be used as initial weights for training both reference networks.

Weights Initialization

As mentioned in Section 3.3.5, pre-trained weights from the Darknet repository are trained on the ImageNet [68] and COCO [69] datasets, which both consist of 3-channel RGB images. The key difference between YOLOv4 with three and a single channel from a structural standpoint is the kernel size of the first convolutional layer. Using the pre-trained weights for three channels on a single-channel network would make it necessary for all following network layers after the first convolutional layer, to adjust for the changed input. This would require an additional training step to adjust the first layer to a single channel before the actual training. Furthermore, finetuning on a single channel of the RGB images would degrade the object detection performance and make it hard to interpret the result of this step. Providing a three-channel input for the network can avoid this. Even though all three infrared channels contain the same information, this should give the infrared network a reasonable starting point for the training. Notice that it is still expected that the infrared reference network is more challenging to train because the used RGB images from pre-training have a different appearance/characteristic than the infrared images. So the LWIR reference network has to adapt to new image characteristics and to images from another domain, while the RGB reference network has only to adapt to images from another domain.

Training

Even with most layers using pre-trained weights in each training run, the convolutional layers for the fusion points are still initialized randomly. This results in slightly different training outcomes, and the evaluation metrics will differ by a certain amount every time. Determining minor differences between different networks can be difficult if the results vary by a small amount. Three individual training runs are conducted on the training server for each reference network to reduce these fluctuations from the training. The best run for each network is chosen as the network to proceed with.

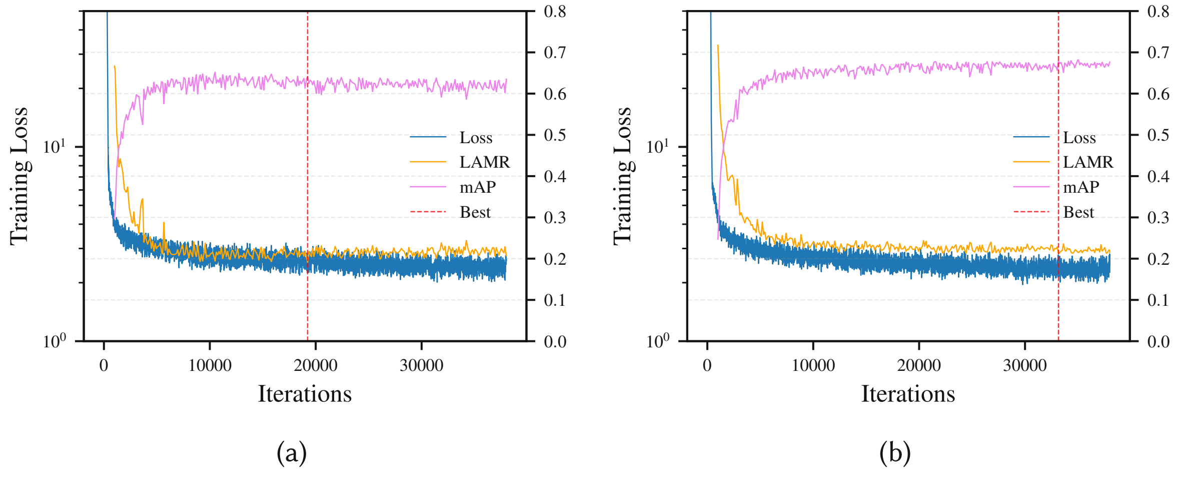


Figure 5.1: Training graph of: (a) RGB reference network; (b) LWIR reference network

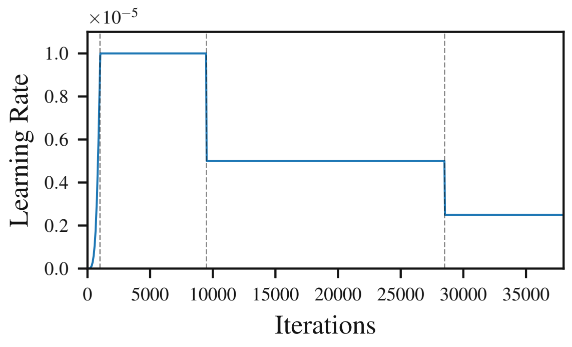


Figure 5.2: Learning rate schedule for the reference training runs

The training runs are carried out with a maximum of over 300 Epochs. This is configured in Darknet via the max_batches parameter and can be calculated with $N_{epochs} = \frac{max\_batches}{batch}$. However, the optimal point is expected to be reached before the 300 epochs. As a learning rate policy, a relatively simple step policy is used, that is derived from testing: The training starts with a learning rate of $1 \cdot 10^{-5}$, and after 9500 batches, the rate is reduced to $0.5 \cdot 10^{-5}$, and finally to $0.25 \cdot 10^{-5}$ after 28500 batches. In addition to the learning rate schedule, the burn-in option is activated: it is used to ramp up the learning from 0 at batch 0 to the learning_rate after burn_in batches with the following equation:

$$lr = lr_{initial} * \left(\frac{iteration}{burn_in}\right)^4 \tag{5.1}$$

This prevents the network parameters from drifting significantly toward the wrong direction at the training begin. This paragraph's discussed parameters are listed in Listing 5.1.

The training charts for the top runs of both reference networks are depicted in Figure 5.1. They include a graph for the training loss, recorded for every epoch, the mAP validation, and the LAMR metrics, both recorded every 4th epoch. In addition, the iteration with the best (lowest) LAMR value is marked with a vertical dotted red line. This checkpoint is the selected checkpoint to continue with. It is used for all further experiments.

Table 5.1: Reference networks training: Best achieved LAMR and mAP metrics

Network	LAMR	Iteration	mAP	Iteration
RGB Reference	19.13	19234	65.20	10502
LWIR Reference	21.24	33158	68.21	35046

Discussion

The training charts (Fig. 5.1) reveal several interesting observations: Firstly, the RGB network achieves its lowest LAMR metric with considerably fewer iterations compared to the LWIR network. This outcome was expected and briefly mentioned when discussing the pre-trained weights for the LWIR network. The RGB network only needs to adjust its parameters during training to handle new RGB images that may have different objects, surroundings, lighting, contrast settings, and field of view. However, the underlying concept of these new RGB images remains the same as the ones in the pretraining datasets. On the other hand, training the LWIR network is more challenging. It not only needs to adapt to the same factors as the RGB network but also to the unique characteristics of the low wavelength infrared spectrum. Another interesting observation relates to the LAMR and mAP metrics as depicted in Table 5.1: It is noticeable that the LWIR network achieves a higher mAP level (better) compared to the RGB network, while the LAMR is slightly higher (worse). This can be attributed to the fact that the mAP metric takes into account all labels in the validation dataset, whereas the LAMR metric focuses only on the labels from the "reasonable" subset and employs logic to ignore irrelevant labels.

5.2 Training of Fusion Networks

After establishing baseline networks for the RGB and LWIR single spectra, this section describes the training of the six different fusion architectures and discusses their results. For the training, several methods and policies are adapted from Section 3. The details of the six fusion architectures are presented in Section 4.2.2. These fusion networks, in contrast to the single spectrum reference networks, take 4-channel (R, G, B, LWIR) PNG images as input.

Network Size

The fusion architectures have the same input image size as the reference networks of 640×640 pixels. This enables a fair comparison with the reference networks.

```
[net]
batch=64
subdivisions=16
width=640
height=640
channels=4
momentum=0.949
decay=0.0005
angle=0
saturation = 1.0
exposure = 1.0
hue=0

learning_rate=0.00001
burn_in=1000
max_batches = 38000
policy=steps
steps=9500,28500
scales=.5,.5

mosaic=1

# Further network definition

# last YOLO layer
[yolo]
random=1
```

Listing 5.2: Darknet config training file for the fusion networks

Augmentation

Similar to the reference networks, augmentations in saturation, hue, and exposure are turned off, and random network size variation and mosaic augmentation are turned on. The used parameters are depicted in the fusion network config in listing 5.2.

Input Channel Interpretation

The most significant difference that can be observed in the fusion networks is the utilization of a 4-channel input image that incorporates both spectra instead of using a single spectrum as in the reference networks. This also comes with structural modifications to the network: The 4-channel input is internally split into two 3-channel images. After that, there are two parallel backbone branches that end up in one or multiple fusion points, after which the backbone regains its original shape. The detailed implementation is described in Section 4.3

Weights Initialization

As discussed in Section 3.3.5, there are two possible configurations for the initial weight initialization per network: one with the remaining main branch of the backbone initialized with pre-trained RB weights and the other with the remaining branch initialized with the LWIR pre-trained weights. Because it is unclear how both variants will influence the results, the training runs are conducted for both variants, and the differences are discussed in Section 5.2.1. Like the reference networks, each training with two configurations was conducted two times, leading to a total of four training runs per network.

Training

To determine the learning rate schedule for the fusion networks, a few short experiments are conducted to examine their usefulness. The results were, that the same learning rate schedule from the reference networks also works fine on the fusion network architectures. Therefore the same configuration that is shown under Figure 5.2 is applied.

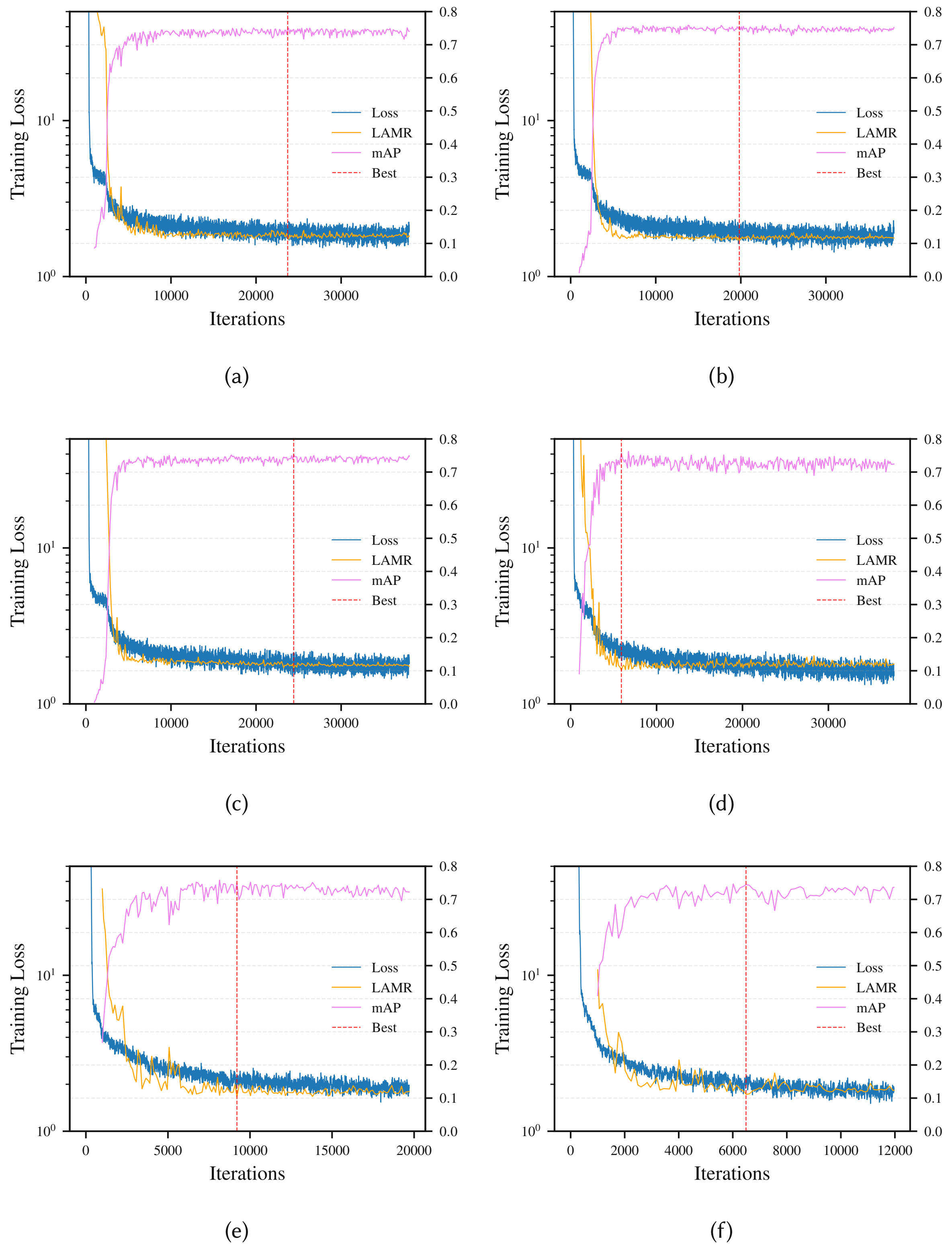


Figure 5.3: Training graphs of fusion architectures: (a) Early Fusion I; (b) Early Fusion II; (c) Mid Fusion I; (d) Mid Fusion II; (e) Late Fusion I; (f) Late Fusion II

Table 5.2: Fusion networks training: Best achieved LAMR and mAP metrics

Network	LAMR	Iteration	mAP	Iteration
Early Fusion I	11.30	23718	75.11	37760
Early Fusion II	10.82	19824	76.14	24662
Mid Fusion I	10.88	24426	75.22	24426
Mid Fusion II	10.14	5900	76.24	6726
Late Fusion I	10.37	9204	75.82	8142
Late Fusion II	11.10	6490	74.66	6018

Discussion

After evaluating the training results, it was decided to stick with the networks trained using the RGB pre-trained weights initialization scheme. Therefore, only the training graphs and discussions related to the RGB pre-trained weight configuration are included in this section. The decision-making process behind this choice and the differences between the two configurations are explained in Section 5.2.1.

Figure 5.3 shows the training graphs of the six fusion architectures. The depicted graphs are taken from the best run of the RGB pre-trained weights initialized network. Each subfigure shows two lines: the training loss over the trained batches and the calculated LAMR for the corresponding batch. Also, the iteration with the best LAMR value is marked with a red dotted line to indicate the selected checkpoint to continue.

When comparing the early fusion networks to the later ones, it becomes apparent that the training process becomes more complicated for the latter due to their increased number of layers and additional feature fusion points for the neck branches. A flattening loss curve for the Early I, Early II, and Mid I before unfreezing the backbone reveals that these fusion networks have already adapted, whereas the ones with later fusion points (Mid II, Late I, Late II) face more challenges. Furthermore, fluctuations in LAMR and mAP indicate a less stable training process overall, likely due to the simultaneous training of multiple branches. Interestingly, architectures with later fusion points tend to reach their best results earlier. An assumption can be made that for the later fusion points, the parallel backbone branches are already trained for their individual spectrum feature extraction, making it comparatively easier to adapt these branches during training.

5.2.1 Initial Weights Influence

An important question is how the two possible weight initialization configurations affect the training results in terms of the metrics. Table 5.3 displays the best LAMR values obtained from each configuration's top-performing run. The depicted LAMR values are calculated on the "reasonable" subset with the detections made by the Darknet FP32 model.

Table 5.3: Fusion architecture weight-initialization: Training differences

Network	RGB	LWIR
Early Fusion I	**11.30**	12.89
Early Fusion II	**10.82**	11.42
Mid Fusion I	**10.88**	11.16
Mid Fusion II	**10.14**	10.48
Late Fusion I	**10.37**	10.57
Late Fusion II	11.10	**10.75**

As seen from Table 5.3, the fusion networks, where the RGB pre-trained weights were used, tend to slightly outperform the LWIR pre-trained approach overall. A possible explanation for this effect is the initialized weights' role in the training process. Even though the network is trained for 300 epochs, the training starts from the given point in the parameter space. Therefore if the network is initialized with the majority of the backbone containing the RGB pre-trained weight, it is expected that the whole network will tend to a solution nearer to the RGB than the LWIR pre-trained weights. The fact that the LAMR metrics are better with the RGB pre-trained weights for all but one fusion architecture leads to the conclusion that the RGB weights adjust more easily to the characteristics of the newly created fused feature maps than the LWIR weights. The RGB pre-trained weights are chosen for further experiments to achieve the best possible results. To notice here is, that the LAMR numbers are calculated for the detections made by the Darknet model during training only. It is expected that these numbers change after the TesnorRT optimízation.

5.2.2 TensorRT Engines

Until now, all networks have been trained using the Darknet framework. In Darknet, these networks consist of a model definition file and a weight file that contain the network parameters. To evaluate the performance of these models, they are converted into TensorRT engines using the workflow described in Section 3.6.1 and shown in Figure 3.5b. This process is implemented through a conversion script that automatically performs all the necessary steps: It extracts layer information from the Darknet model definition with the help of a Darknet fork and builds the TensorRT engines from it with the tkDNN library. The conversion process is essential for ensuring the models can be efficiently used in real-world applications on the embedded NVIDIA device.

TensorRT provides the capability of using different precisions for internal operations. TensorRT calculates internal operations in the Floatingpoint32 (FP32) datatype by default. However, other precisions, such as Floatingpoint16 (FP16 or Half precision) and Integer8 (Int8) bit, are also possible. Additionally, TensorRT supports automatic mixed-precision mode (best mode), which automatically chooses a datatype for each layer to balance the network optimally. Using different datatypes for internal precision can significantly impact the latency of neural networks, with lower precision datatypes typically resulting in faster inference times. However, the trade-off is a potential decrease in object detection accuracy. Therefore, it is crucial to carefully evaluate the impact of different precision modes on the inference speed and model accuracy before selecting the optimal datatype for a given application.

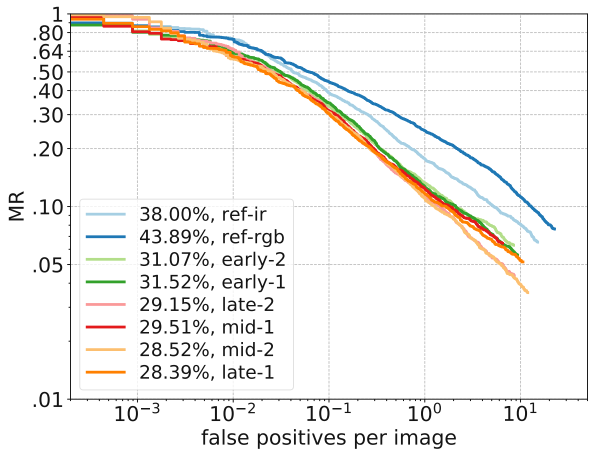
From personal experience, using FP16 precision for internal operations gives a significant boost in latency while only losing minimal accuracy. However, the impact of using FP16 precision depends on the specific network being used. All Darknet models are converted to TensorRT engines with FP16 precision for the experiments. For case-by-case comparisons, the FP32 engines are also generated. In section 5.3, there are some comparisons to study the possible degradation of LAMR caused by the TensorRT precision change.

5.3 Object Detection Performance

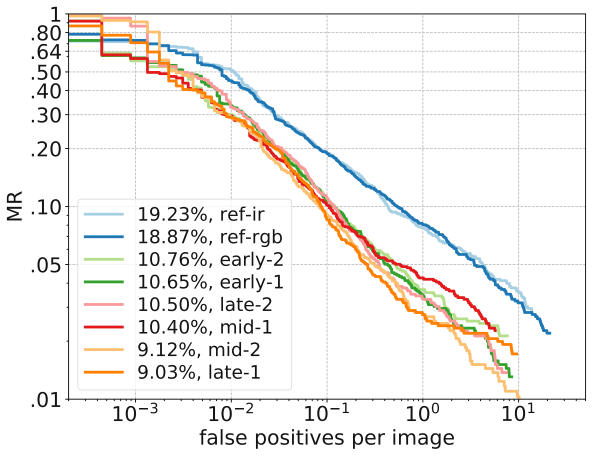
A key point why multispectral feature fusion is done in the first place is the possible improvement of the object detection performance. The goal is that the fusion architectures should be able better in detecting the desired objects than the reference networks. While these fusion networks should perform better than the reference networks in adverse weather, these experiments only concentrate on the general case with undistorted spectra. Several metrics provide a measure for this. The most often used ones are mAP and LAMR as discussed in Section 2.1.1. The primary metric chosen for the experiments is LAMR. For every architecture and reference network, the miss rate curves are recorded and depicted in Figure 5.4 and the resulting LAMR values are listed in Table 5.4. This includes the “all” and the “reasonable” subsets from the KAIST dataset. In addition, both subsets are split into day and night scenes to study the effects on day scenes, where the assumption is that the RGB images will provide more information, and the night scenes, where the assumption is that that the LWIR images will provide more information. These experiments were conducted with the optimized TensorRT engines with an internal precision of FP16.

Discussion

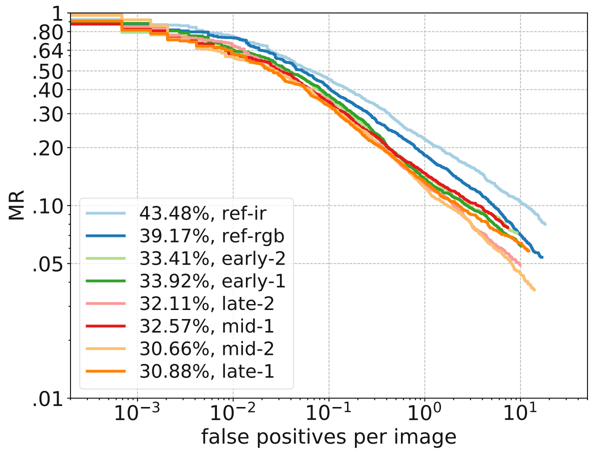
These graphs are interpreted as follows: The area under the curve corresponds to the LAMR value, whereas a smaller area indicates a lower LAMR. The LAMR value is derived from the average of nine logarithmically equidistant points between 10^{-2} and 10^{0} on the x-axis. Intuitively, this can be understood as follows: A lower LAMR corresponds to a lower miss rate for a given number of false positives



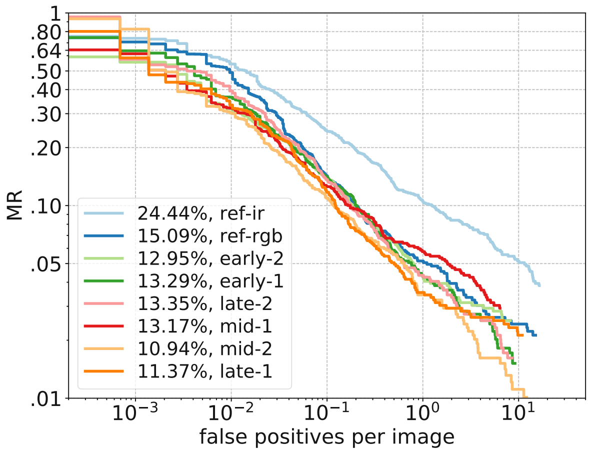
(a)



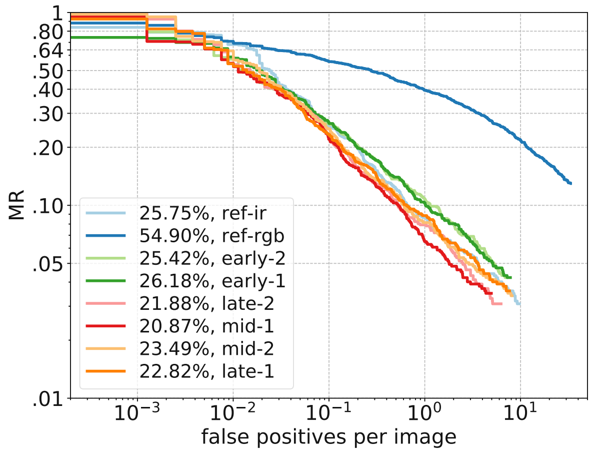
(b)



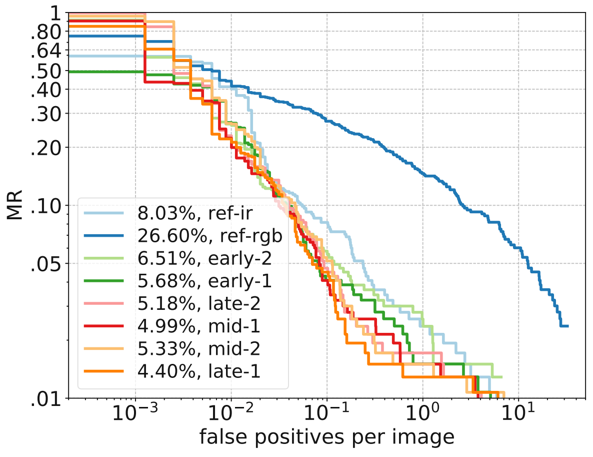
(c)



(d)



(e)



(f)

Figure 5.4: Object detection performance of FP16 TensorRT fusion architectures: (a) "all" (day+night); (b) "reasonable" (day+night); (c) "all" (day); (d) "reasonable" (day); (e) "all" (night); (f) "reasonable" (night);

per image.

The first row of Figure 5.4 show the miss rate curves on the “all” (Subfig. 5.4a) and the “reasonable” subset (Subfig. 5.4b). Both subsets contain images from the day and night scenes. The first thing to notice here is that all fusion architectures provide better LAMR values than the reference networks: At first glance, the LAMR numbers of the fusion architectures lie relatively close together in absolute numbers, but they are all significantly lower than the reference numbers. Between the best-performing reference network (LWIR reference) and the best-performing fusion architecture (Late I) on the “all” subset is a difference of 9.61%. On the “reasonable” subset, there is a difference of 9.84% between the best-performing reference net (RGB) and the best-performing fusion network (Late I). The best LAMR value in both subsets is achieved with the Late I network. For the reference networks, the LWIR network outperforms the RGB network on the “all” subset, while the RGB is better on the “reasonable” subset.

Comparing the reference networks in night and day scenes in general also leads to the conclusion that the RGB images work better on the day scenes (Subfigs. 5.4c and 5.4d) than the LWIR images and vice versa on the night scenes (Subfigs. 5.4e and 5.4f). A key difference between them is that while the LWIR reference network performs moderately worse than the RGB network in day scenes (43.48% vs. 39.17% on the “all” subset), the RGB network performs significantly worse than the LWIR in night scenes (54.90% vs. 25.75% on the “all” subset). This significant degradation of the RGB network in night scenes is also likely why the LWIR reference network performs better than the RGB network overall on the day+night scenes (38.00% vs. 43.89% on the “all” subset).

A potential explanation for the slight superiority of the RGB network on the “reasonable” subset could be observed when examining the day (Subfig. 5.4d) and night (Subfig. 5.4f) subsets. While the degradation of RGB images in night scenes remains evident, it is also notable that the RGB network outperforms the LWIR network in day scenes by a significant margin. When considering the overall performance, this results in the RGB reference network being slightly superior on the “reasonable” day+night subset (Subfig. 5.4b).

A bar chart with the LAMR values is depicted in Figure 5.5. The LAMR values are for the “all” and for the “reasonable” subset. An interesting observation is regarding the trend between the architecture and the LAMR values: The chart indicates a local minimum around the Mid Fusion II and the Late Fusion I fusion positions. This observation is consistent with literature like Feng et al. [47], where the authors describe the mid-fusion approaches as a good compromise of exploiting the features between spatial resolution and feature dimension.

Network	Reasonable				All			
	All	Day	Night	Recall	All	Day	Night	Recall
RGB Reference	0.189	0.151	0.266	0.978	0.439	0.392	0.549	0.924
IR Reference	0.192	0.244	0.080	0.971	0.380	0.435	0.257	0.935
Early Fusion I	0.106	0.133	0.057	0.987	0.315	0.339	0.262	0.944
Early Fusion II	0.108	0.130	0.065	0.979	0.311	0.334	0.254	0.937
Mid Fusion I	0.104	0.132	0.050	0.977	0.295	0.326	0.209	0.936
Mid Fusion II	0.091	0.109	0.053	0.990	0.285	0.307	0.235	0.964
Late Fusion I	0.090	0.114	0.044	0.983	0.284	0.309	0.228	0.948
Late Fusion II	0.105	0.134	0.052	0.986	0.291	0.321	0.219	0.956

Table 5.4: LAMR results for TensorRT FP16 networks.

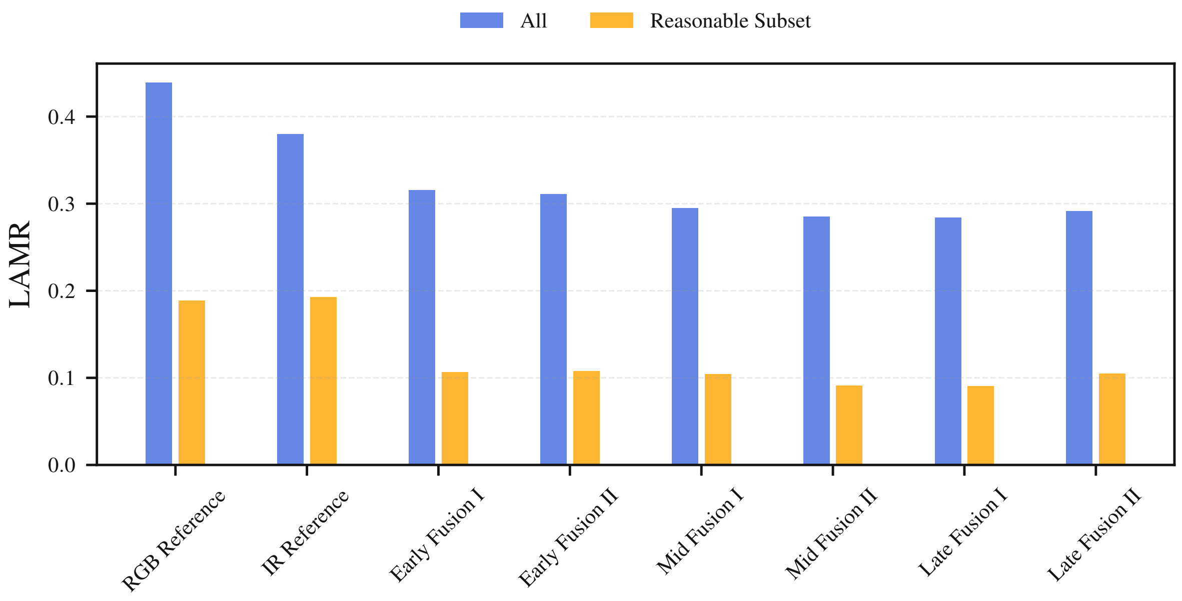


Figure 5.5: Object detection performance: LAMR on KAIST “reasonable” subset

5.4 Network Latency

When deploying a neural network on an embedded device, latency is a crucial factor to consider. Analyzing how fusion architectures impact this metric is essential if a target latency is specified for the network. This experiment measures the network latency for the platform-unoptimized Darknet and platform-optimized TensorRT models. The goal is to study the performance gains by TensorRT through different internal precisions of the model and study the degradation between the reference networks (that will have the lowest latency) and the feature fusion architectures (that will have an increased latency due to additional network operations). The inference times are measured under specific system conditions (Sec. 3.5.3). In these conditions, the only running application is the network inference, no power target (unlimited) is set, and the DVFS governer is disabled.

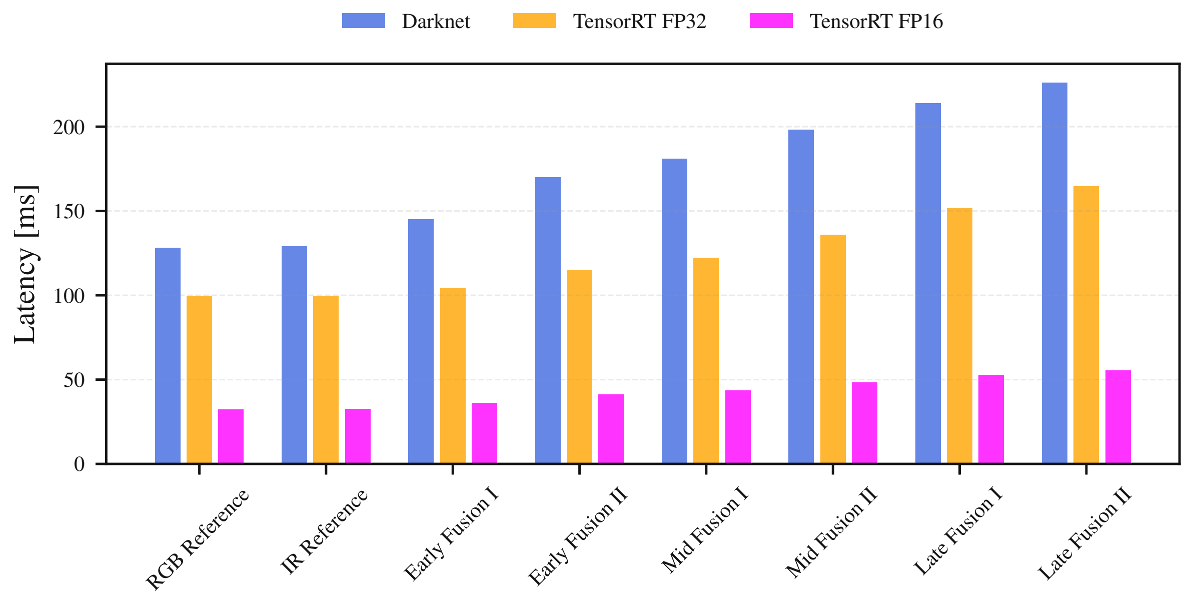


Figure 5.6: Network latencies of different optimized models

First, the baseline is introduced with the Darknet models. These models can be considered not platform-optimized. However, they can be considered hardware optimized because the Darknet was compiled with CUDA and cuDNN support. This hardware optimization is generic and can not use specific implemented features like the Jetson platform. Furthermore, the optimized TensorRT engines with internal model precisions of FP32 and FP16 are measured. These engines are optimized for the test platform (Jetson AGX Xavier) and are expected to gain a significant speed advantage compared to the Darknet models. Table 5.5 presents the values obtained from the measurements, which are also visually represented in Figure 5.6.

Network	Darknet [ms]	TensorRT FP32 [ms]	TensorRT FP16 [ms]
RGB Reference	128.00	99.19	32.29
IR Reference	129.00	99.22	32.50
Early Fusion I	145.00	104.13	36.06
Early Fusion II	170.00	114.91	41.18
Mid Fusion I	181.00	122.28	43.48
Mid Fusion II	198.00	135.72	48.18
Late Fusion I	214.00	151.44	52.82
Late Fusion II	226.00	164.54	55.49

Table 5.5: Latency measurements from all network architectures

Discussion

The measured latency values for the two TensorRT engines and the Darknet model are graphically depicted in Figure 5.6 and in textual form in Table 5.5. Several observations can be made from the measurement results. Firstly, when comparing the three model types based on absolute numbers, it is evident that the Darknet model, which utilizes FP32 operations internally, has the highest latency among all models. This outcome is expected because, as mentioned earlier, Darknet relies on generic optimizations from CUDA and CUDNN without leveraging platform-specific optimizations as TensorRT does. Consequently, the darknet model is approximately 30% slower than the TensorRT FP32 model for the RGB Reference network and about 37% slower for the Late II fusion architecture.

Another comparison can be made between the TensorRT FP32 and TensorRT FP16 models. Across all models, it is noteworthy that the FP16 models demonstrate a significantly faster performance, approximately three times faster, compared to the FP32 models. This emphasizes the significant improvement in latency that can be achieved by utilizing FP16 (half precision) operations on the Jetson platform.

When discussing the impact of feature fusion architectures, it is crucial to consider the comparison between the reference networks and the fusion architectures. Focusing on the FP16 TensorRT model, the reference models exhibit latencies of 32.29 ms (RGB) and 32.50 ms (LWIR). In comparison, the slowest network, the Late II architecture, is 71.8% slower, while the fastest fusion architecture, the Early 1, is 11.7% slower than the RGB reference network. It is important to note that these increases in latency must be evaluated in relation to the improvements in LAMR achieved by the fusion architectures and cannot be assessed in isolation.

5.5 Complexity-Latency Correlation

Network complexity plays a significant role in better understanding the source of latency increases in fusion architectures. Put simply, when additional network operations are required in a resource-constrained system, the latency increases almost always because more time is needed to process the increased number of operations. In other words, the more complex the network becomes, the longer it takes to perform the necessary computations, resulting in higher latency.

A question arises regarding the correlation between the number of network operations and the latency after optimizing the network with TensorRT. It is unclear whether TensorRT can apply optimization techniques that help reduce the required operations between the parallel backbone branches. To investigate this potential effect, the correlation coefficients between the number of operations in Darknet models and the latency values of Darknet, as well as both TensorRT engines, are calculated. This analysis aims to determine the TensorRT optimization capabilities for the proposed feature fusion architectures. To investigate this, the Pearson correlation coefficient between network complexity and latencies is calculated for the Darknet, TensorRT FP16, and TensorRT FP32 models with the following equation:

$$r = \frac{\sum (x - \overline{x})(y - \overline{y})}{\sqrt{\sum (x - \overline{x})^2 \sum (y - \overline{y})^2}} \tag{5.2}$$

The network complexities are parsed from the Darknet log and listed in Table 5.6. The Pearson correlation coefficients are calculated with Equation 5.2 and depicted in Table 5.7. Furthermore, for better visualization of the data points, they are also shown in a scatterplot for all three models in Figure 5.7 with linear approximated trendlines.

Network	FLOPS (in billion)
RGB Reference	140.98
IR Reference	140.98
Early Fusion I	143.36
Early Fusion II	155.53
Mid Fusion I	165.39
Mid Fusion II	187.84
Late Fusion I	211.75
Late Fusion II	227.48

Table 5.6: Network complexity of all architectures

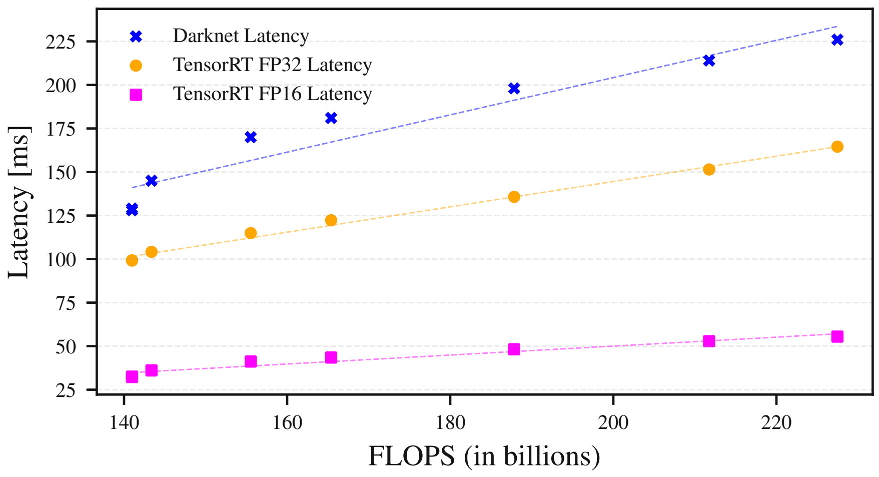


Figure 5.7: Network complexity visualization

Table 5.7: Pearson correlation between network complexity and latency

	Darknet Latency	TensorRT FP32 Latency	TensorRT FP16 Latency
FLOPS	0.959	0.996	0.974

Discussion

The correlation coefficients in Table 5.7 are all close to 1.0, indicating a strong positive relationship between network complexity and latencies. If TensorRT could optimize the parallel backbone branches to a certain degree, this correlation would likely be weaker. This observation is also evident in Figure 5.7, which illustrates a strong linear relationship between Flops and Latency.

5.6 Power Consumption

Another important measurement of an embedded device is the power consumption a network inference has. Devices like the Jetsons can be configured with power targets to restrict system power or set to an unlimited target. Along with the power target, various system clocks, CPU core configurations, and other parameters are adjusted.

This experiment aims to investigate the impact of different network architectures on the Jetson AGX Xavier. An important question is whether the GPU is already fully utilized by the reference networks or if there are additional resources that become engaged as network complexity increases in fusion architectures. To conduct this experiment, the methodology described in Section 3.5.2 is employed. The power target is set to unlimited, ensuring that all clocks operate at maximum speed, and disable the DVFS to avoid running into an artificial power target. The network inference is kept running in a continuous loop for 10 minutes and the average power consumption is recorded. The measured average power consumption values are listed in Table 5.8 and visualized in Figure 5.8.

Network	GPU Power [W]	GPU Energy[1] [J]	Total Power [W]	Total Energy[1] [J]
RGB Reference	28.07	0.91	39.71	1.28
IR Reference	27.19	0.88	38.91	1.26
Early Fusion I	27.52	0.99	39.33	1.42
Early Fusion II	27.99	1.15	40.24	1.66
Mid Fusion I	27.86	1.21	40.45	1.76
Mid Fusion II	28.06	1.35	40.88	1.97
Late Fusion I	27.82	1.47	41.25	2.18
Late Fusion II	27.83	1.54	41.41	2.30

[1] Energy per network inference

Table 5.8: Average power consumption of all network architectures

Discussion

When examining the recorded data, two noticeable things can be observed. Firstly, there appears to be no clear trend in the average GPU power consumption concerning the fusion architecture. Secondly, a slight upward trend can be observed in the total system power, which appears to correlate with the position of the fusion point.

The nearly constant values of GPU power consumption across all networks serve as an indicator that the GPU is fully utilized for each network. In other words, all available GPU resources are being utilized.

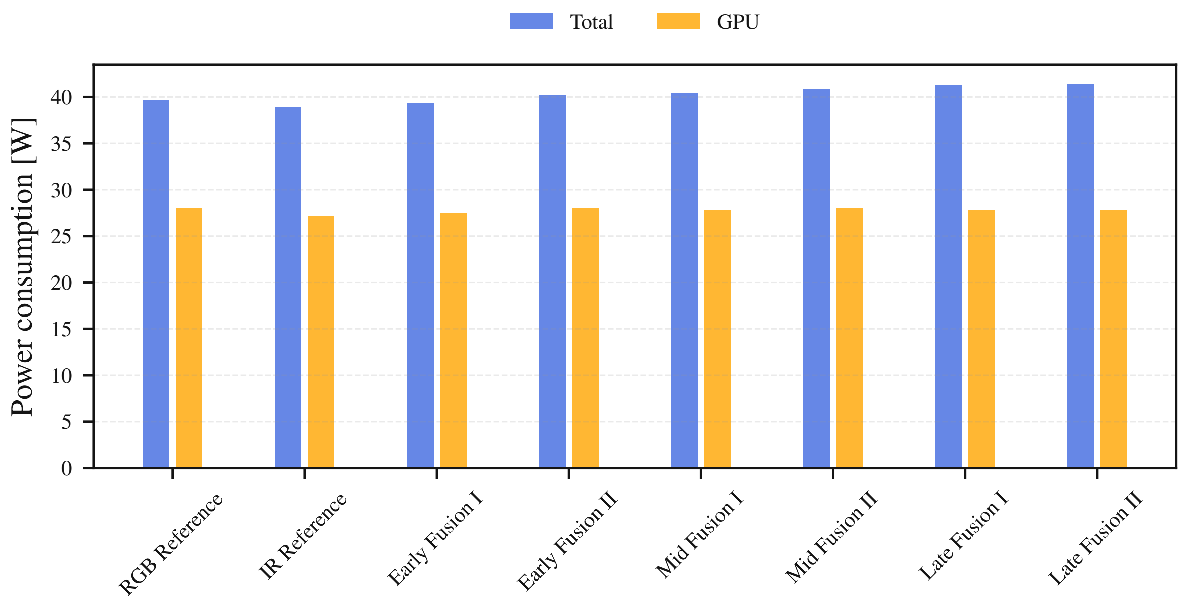


Figure 5.8: Average power consumption of all network architectures

This observation leads to the assumption that the observable impact of higher network complexity is primarily reflected in network latency rather than GPU power consumption. Another observation pertains to the overall system power consumption, which exhibits a slight upward trend corresponding to the position of the fusion point. I assume that this trend is attributed to increased memory activity resulting from the larger number of parameters and kernel switches required for the fusion networks. However, the largest observed difference in total power consumption between the LWIR reference network with a value of 38.908 W and the Late Fusion II network with a value of 41.406 W is relatively small in absolute numbers. Therefore, this difference is likely to play a minor role as a selection criterion.

5.7 Discussion

In this section, a series of experiments are conducted to study the impact of TensorRT optimization on the object detection performance of both the reference and fusion networks. Furthermore, the experiments investigate how feature fusion architectures affect latency and power consumption. All experiments are performed on the Jetson AGX Xavier platform, widely recognized as an industry-standard embedded device for machine learning applications.

However, the results of these experiments are discussed in isolation, and a question remains: How can these individual experiment findings be combined to evaluate and select an architecture for an applicable solution?

5.7.1 Application Viability

Determining whether it makes sense to use a multispectral feature fusion architecture for a particular use case is always an individual decision. It depends on various factors, including the specific problem one aims to solve with such an approach. For certain use cases, the enhanced object detection performance offered by fusion architectures may be the most important advantage, with latency and power consumption considered secondary. Generally, the design decision of whether to utilize a fusion architecture and, if so, which specific architecture depends on multiple factors, as explored in the conducted experiments. Evaluating the improvement in object detection performance against the baseline can be done in comparison to the increase in latency and power consumption. Often, selecting a specific network will be a tradeoff between these factors.

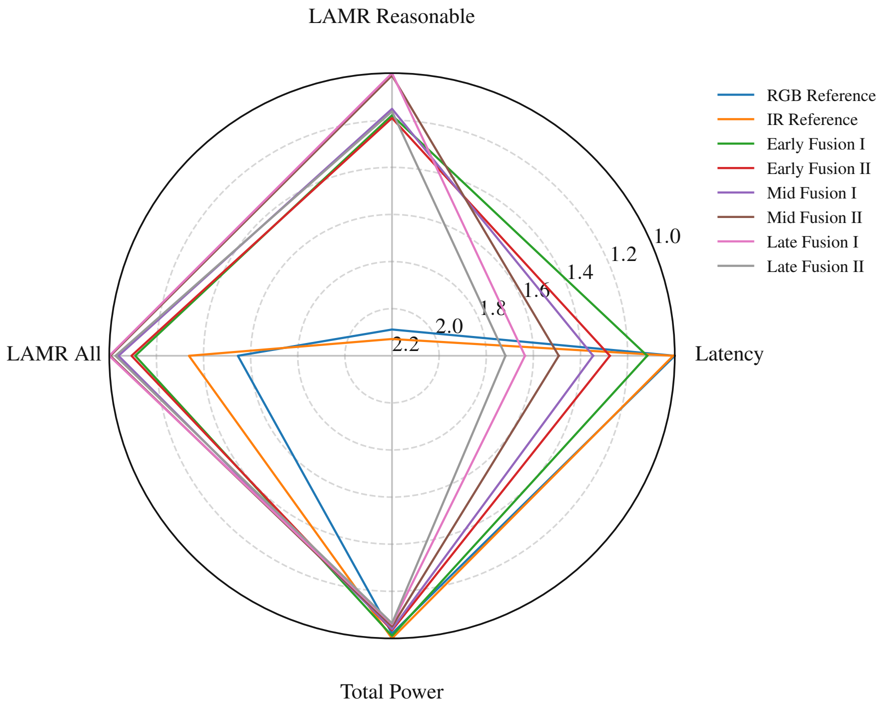


Figure 5.9: Optimization factors of fusion architectures

One possible approach to weighing the tradeoffs between multiple factors is by using a radar plot. A radar plot (also known as a spider chart), is a graphical representation that allows for the visualization and comparison of multiple variables simultaneously. By plotting different factors on the axes of the radar plot, the relative importance or performance of each factor can be assessed and compared, helping in the decision-making process. Based on the factors studied in this work, a radar plot can be created to include the following metrics: object detection performance (LAMR metric for both the "all" and "reasonable" subsets), latency, and system power consumption.

A possible interpretation of the described radar plot is depicted in Figure 5.9. Each axis represents a factor associated with choosing a particular design. It's important to note that the graphical results of a radar plot depend on how the values of the factors are preprocessed. In this figure, is is chosen to divide all values on each axis by the smallest achieved number. This approach results in a scale that depicts the increase of the factors relative to the best-achieved value. However, it's important to highlight that the values are not fully normalized, as doing so could potentially detract from the importance of certain factors, such as total power consumption.

An approach to selecting a design based on the radar plot is to consider the area size that a design covers. In this particular radar plot, the RGB and LWIR reference networks exhibit the best values in terms of latency and total power factors. However, they show relatively lower performance on the object detection scale.

An approach that covers the largest area and provides a balanced tradeoff between all factors in the radar plot would be the Early Fusion I approach. This approach achieves significant reductions in LAMR, specifically by 44% on the "reasonable" subset and 28% on the "all" subset, while only introducing a slight increase of 12% in network latency. By considering these metrics together, Early Fusion I demonstrates a favorable tradeoff from a practical standpoint.

5.7.2 Challenges and Limitations

It is worth noting that the depicted radar plot and the experiments conducted in this work have certain limitations. Two critical factors implicitly assumed in the experiments are the spatial alignment of the multispectral image channels and the availability of all spectra.

As briefly discussed in Section 4.1.3, spatial alignment refers to the precise alignment of different channels within a multispectral image. It plays a crucial role in accurate feature extraction and fusion. Additionally, the availability of all spectra refers to the inclusion of all relevant spectral bands in the analysis. The possibility of spectra dropping out due to errors in camera sensors raises an important question regarding the influence on the object detection performance of the system. When certain spectra are missing or not captured accurately, it can impact the system's ability to detect and classify objects effectively.

These questions are crucial to address before implementing fusion architectures in real-world use cases. They are essential for the overall system design, as they significantly impact the effectiveness and functionality of the architecture in practical applications. Therefore, it is important to note that the results and conclusions of the experiments conducted in this thesis were based on data where spatial alignment and spectra dropout were not considered. It is highly probable that if these challenges were taken into account, the outcomes and conclusions of the experiments would be different.

Chapter 6

Conclusion and Future Outlook

This thesis aimed to investigate the application of multispectral feature fusion architectures on an embedded hardware platform. The focus was on studying the impact of these feature fusion architectures' on hardware because these types of experiments are rarely conducted in other work. This thesis's methodology for selecting and training feature fusion approaches focused on practicality and applicability. As a result, the chosen methods were kept straightforward and uncomplicated.

First, YOLOv4 was selected as the base object detection architecture to do further studies. At the beginning of this thesis, it could be considered a SOTA one-stage object detector. Due to the large design space a possible feature fusion approach has in a given network, the focus was on backbone fusion with an easy-to-implement fusion scheme. The primary design variable left was the fusion position in the backbone. The KAIST dataset, which contains RGB visible light images, and LWIR thermal images, was chosen for the experiments. A specialty of this dataset is the spatial alignment of the spectra channels, so the challenge of spatially aligning different image sources could be neglected in the experiments.

Two reference and six feature fusion networks are selected and trained with the Darknet framework afterward. These networks were optimized for the target platform in this work: a Jetson AGX Xavier. On this optimized network, multiple experiments are conducted. It was studied how the pre-trained weights initialization scheme could potentially influence the object detection performance. The object detection performance was measured with the metric of LAMR, showing that all feature fusion approaches performed significantly better than the single-spectrum reference networks. This is true for the easy-to-detect objects in the "reasonable" subset and the more realistic 'all' set. Latency experiments showed a strong correlation between network complexity and network latency. This was true for the unoptimized Darknet models and the platform-optimized TensorRT models. Power consumption experiments on the Jetson Xavier showed that for the studied networks, the GPU capacity is fully utilized, and therefore the only metric to observe the increased network operations is the network latency. In the end, a way has been shown how such a design selection can be made. From a practical perspective in this thesis' experimental setup, the Early Fusion I approach reduced the achieved LAMR on the "reasonable" subset by 44% and by 28% on the "all" subset against the RGB reference network, while only causing an increase of 12% in network latency. In terms of applicability, such a fusion approach could be considered beneficial when the increase in latency is acceptable.

For future work, two design aspects neglected in the experimental setup are of great interest: The spatial alignment between the single spectra and the robustness of channel dropouts. How spatial alignment influences object detection performance regarding the fusion position would be of great interest. Also, how robust the fusion architectures are if a single spectrum cuts out. The goal would be that it then performs not worse than the reference network, but this seems a complex problem.

In conclusion, this thesis sheds light on the influence of feature fusion architectures when optimized using TensorRT for the NVIDIA Jetson platform. While it is not possible to make a general statement, specific findings from this work could serve as valuable insights for future design considerations.

Bibliography

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp. 770–778, iSSN: 1063-6919.

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[3] H. Iqbal, "HarisIqbal88/PlotNeuralNet v1.0.0," Dec. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.2526396

[4] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A Fast Learning Algorithm for Deep Belief Nets," *Neural Computation*, vol. 18, no. 7, pp. 1527–1554, Jul. 2006.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://papers.nips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

[6] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-December. IEEE Computer Society, Dec. 2016, pp. 779–788, arXiv: 1506.02640 ISSN: 10636919.

[7] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "YOLOv4: Optimal Speed and Accuracy of Object Detection," Apr. 2020, arXiv: 2004.10934. [Online]. Available: http://arxiv.org/abs/2004.10934

[8] S. Hwang, J. Park, N. Kim, Y. Choi, and I. S. Kweon, "Multispectral pedestrian detection: Benchmark dataset and baseline," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Jun. 2015, pp. 1037–1045. [Online]. Available: http://ieeexplore.ieee.org/document/7298706/

[9] H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid, and S. Savarese, "Generalized intersection over union: A metric and a loss for bounding box regression," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2019-June. IEEE Computer Society, Jun. 2019, pp. 658–666, arXiv: 1902.09630 ISSN: 10636919.

[10] Z. Zheng, P. Wang, W. Liu, J. Li, R. Ye, and D. Ren, "Distance-IoU loss: Faster and better learning for bounding box regression," in *AAAI 2020 - 34th AAAI Conference on Artificial Intelligence*. AAAI press, 2020, pp. 12 993–13 000, arXiv: 1911.08287 ISSN: 2159-5399.

[11] M. Everingham, S. M. 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, no. 1, pp. 98–136, Jan. 2015, publisher: Kluwer Academic Publishers.

[12] P. Dollár, C. Wojek, B. Schiele, and P. Perona, “Pedestrian detection: An evaluation of the state of the art,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 4, pp. 743–761, 2012.

[13] ——, “Pedestrian detection: A benchmark,” in *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, Jun. 2009, pp. 304–311. [Online]. Available: https://ieeexplore.ieee.org/document/5206631/

[14] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, Dec. 1943.

[15] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain.” *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958.

[16] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[17] M. Minsky and S. A. Papert, *Perceptrons: an introduction to computational geometry*, 2nd ed. Cambridge/Mass.: The MIT Press, 1972.

[18] D. O. Hebb, *The organization of behavior; a neuropsychological theory.*, ser. The organization of behavior; a neuropsychological theory. Oxford, England: Wiley, 1949, pages: xix, 335.

[19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning internal representations by error propagation,” in *Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1: Foundations*. Cambridge, MA, USA: MIT Press, 1986, pp. 318–362, number of pages: 45.

[20] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. O’Reilly Media, Inc, 2019.

[21] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, ser. Proceedings of machine learning research, G. Gordon, D. Dunson, and M. Dudík, Eds., vol. 15. Fort Lauderdale, FL, USA: PMLR, Apr. 2011, pp. 315–323, tex.pdf: http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf. [Online]. Available: https://proceedings.mlr.press/v15/glorot11a.html

[22] K. P. Murphy, *Probabilistic Machine Learning: An introduction*. MIT Press, 2022. [Online]. Available: probml.ai

[23] L. Lu, Y. Shin, Y. Su, and G. Karniadakis, “Dying ReLU and Initialization: Theory and Numerical Examples,” *Communications in Computational Physics*, vol. 28, pp. 1671–1706, Nov. 2020.

[24] S. Ioffe and C. Szegedy, “Batch normalization: accelerating deep network training by reducing internal covariate shift,” in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML’15. Lille, France: JMLR.org, Jul. 2015, pp. 448–456.

[25] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: Data mining, inference, and prediction*, ser. Springer series in statistics. Springer New York, 2013, tex.lccn: 2001031433. [Online]. Available: https://books.google.at/books?id=yPfZBwAAQBAJ

[26] B. Polyak, “Some methods of speeding up the convergence of iteration methods,” *Ussr Computational Mathematics and Mathematical Physics*, vol. 4, pp. 1–17, Dec. 1964.

[27] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, “On the importance of initialization and momentum in deep learning,” *30th International Conference on Machine Learning, ICML 2013*, pp. 1139–1147, Jan. 2013.

[28] J. Duchi and E. Hazan, “Adaptive Subgradient Methods for Online Learning and Stochastic Opti-

mization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, Jul. 2011.

[29] T. Tieleman, G. Hinton, and others, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[30] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *International Conference on Learning Representations*, Dec. 2014.

[31] Y. Nesterov, "A method for solving the convex programming problem with convergence rate O(1/k^2)," *Proceedings of the USSR Academy of Sciences*, 1983. [Online]. Available: https://www.semanticscholar.org/paper/A-method-for-solving-the-convex-programming-problem-Nesterov/8d3a318b62d2e970122da35b2a2e70a5d12cc16f

[32] I. Loshchilov and F. Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts," *arXiv: Learning*, Aug. 2016. [Online]. Available: https://www.semanticscholar.org/paper/SGDR%3A-Stochastic-Gradient-Descent-with-Warm-Loshchilov-Hutter/b022f2a277a4bf5f42382e86e4380b96340b9e86

[33] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, Dec. 1989, conference Name: Neural Computation.

[34] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998, conference Name: Proceedings of the IEEE.

[35] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, Sep. 2014, pp. 580–587, arXiv: 1311.2524 ISSN: 10636919.

[36] R. Girshick, "Fast R-CNN," in *2015 IEEE International Conference on Computer Vision (ICCV)*, Dec. 2015, pp. 1440–1448, iSSN: 2380-7504.

[37] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, pp. 1137–1149, Jun. 2017, arXiv: 1506.01497 Publisher: IEEE Computer Society.

[38] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," Dec. 2016, arXiv: 1612.08242. [Online]. Available: http://arxiv.org/abs/1612.08242

[39] ——, "YOLOv3: An Incremental Improvement," Apr. 2018, arXiv: 1804.02767. [Online]. Available: http://arxiv.org/abs/1804.02767

[40] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single Shot MultiBox Detector," in *Computer Vision – ECCV 2016*, ser. Lecture Notes in Computer Science, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 21–37.

[41] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results." [Online]. Available: http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html

[42] G. Jocher, A. Chaurasia, A. Stoken, J. Borovec, NanoCode012, Y. Kwon, K. Michael, TaoXie, J. Fang, Imyhxy, Lorna, Z. Yifu, C. Wong, A. V, D. Montes, Z. Wang, C. Fati, J. Nadar, Laughing, UnglvKitDe, V. Sonck, Tkianai, YxNONG, P. Skalski, A. Hogan, D. Nair, M. Strobel, and M. Jain, "ultralytics/yolov5: v7.0 - YOLOv5 SOTA Realtime Instance Segmentation," Nov. 2022. [Online]. Available: https://zenodo.org/record/3908559

[43] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, “YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors,” Jul. 2022, arXiv: 2207.02696. [Online]. Available: http://arxiv.org/abs/2207.02696

[44] C. Y. Wang, H. Y. Mark Liao, Y. H. Wu, P. Y. Chen, J. W. Hsieh, and I. H. Yeh, “CSPNet: A new backbone that can enhance learning capability of CNN,” in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, vol. 2020-June. IEEE Computer Society, Jun. 2020, pp. 1571–1580, arXiv: 1911.11929 ISSN: 21607516.

[45] K. He, X. Zhang, S. Ren, and J. Sun, “Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 9, pp. 1904–1916, Sep. 2015, publisher: IEEE Computer Society.

[46] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, “Path Aggregation Network for Instance Segmentation,” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, Dec. 2018, pp. 8759–8768, arXiv: 1803.01534 ISSN: 10636919.

[47] D. Feng, C. Haase-Schutz, L. Rosenbaum, H. Hertlein, C. Glaser, F. Timm, W. Wiesbeck, and K. Dietmayer, “Deep Multi-Modal Object Detection and Semantic Segmentation for Autonomous Driving: Datasets, Methods, and Challenges,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 3, pp. 1341–1360, Mar. 2021, arXiv: 1902.07830 Publisher: Institute of Electrical and Electronics Engineers Inc.

[48] R. Gade and T. B. Moeslund, “Thermal cameras and applications: A survey,” *Machine Vision and Applications*, vol. 25, no. 1, pp. 245–262, Jan. 2014.

[49] J. Liu, S. Zhang, S. Wang, and D. N. Metaxas, “Multispectral deep neural networks for pedestrian detection,” in *British Machine Vision Conference 2016, BMVC 2016*, vol. 2016-September. British Machine Vision Conference, BMVC, 2016, pp. 73.1–73.13, arXiv: 1611.02644.

[50] L. Zhang, X. Zhu, X. Chen, X. Yang, Z. Lei, and Z. Liu, “Weakly Aligned Cross-Modal Learning for Multispectral Pedestrian Detection,” in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE, Oct. 2019, pp. 5126–5136. [Online]. Available: https://ieeexplore.ieee.org/document/9009100/

[51] H. Zhang, E. Fromont, S. Lefevre, and B. Avignon, “Multispectral Fusion for Object Detection with Cyclic Fuse-and-Refine Blocks,” in *2020 IEEE International Conference on Image Processing (ICIP)*. IEEE, Oct. 2020, pp. 276–280. [Online]. Available: https://ieeexplore.ieee.org/document/9191080/

[52] “Teledyne FLIR ADAS Dataset.” [Online]. Available: https://www.flir.com/oem/adas/adas-dataset-form/

[53] Y. Zhang, Z. Yin, L. Nie, and S. Huang, “Attention Based Multi-Layer Fusion of Multispectral Images for Pedestrian Detection,” *IEEE Access*, vol. 8, pp. 165 071–165 084, 2020, publisher: Institute of Electrical and Electronics Engineers Inc. [Online]. Available: https://ieeexplore.ieee.org/document/9187824/

[54] A. Wolpert, M. Teutsch, M. S. Sarfraz, and R. Stiefelhagen, “Anchor-free Small-scale Multispectral Pedestrian Detection,” Aug. 2020, arXiv: 2008.08418. [Online]. Available: http://arxiv.org/abs/2008.08418

[55] W. Liu, I. Hasan, and S. Liao, “Center and Scale Prediction: Anchor-free Approach for Pedestrian and Face Detection,” Apr. 2019, arXiv: 1904.02948. [Online]. Available: http://arxiv.org/abs/1904.02948

[56] C. Li, D. Song, R. Tong, and M. Tang, “Multispectral Pedestrian Detection via Simultaneous Detection and Segmentation,” Aug. 2018, arXiv: 1808.04818. [Online]. Available: http://arxiv.org/abs/1808.04818

[57] F. Farahnakian and J. Heikkonen, "Deep learning based multi-modal fusion architectures for maritime vessel detection," *Remote Sensing*, vol. 12, no. 16, Aug. 2020, publisher: MDPI AG.

[58] Z. Cao, H. Yang, J. Zhao, S. Guo, and L. Li, "Attention fusion for one-stage multispectral pedestrian detection," *Sensors*, vol. 21, no. 12, Jun. 2021, publisher: MDPI AG.

[59] T. Karasawa, K. Watanabe, Q. Ha, A. Tejero-De-Pablos, Y. Ushiku, and T. Harada, "Multispectral object detection for autonomous vehicles," in *Thematic Workshops 2017 - Proceedings of the Thematic Workshops of ACM Multimedia 2017, co-located with MM 2017.* Association for Computing Machinery, Inc, Oct. 2017, pp. 35–43.

[60] J. Nataprawira, Y. Gu, I. Goncharenko, and S. Kamijo, "Pedestrian detection using multispectral images and a deep neural network," *Sensors*, vol. 21, no. 7, Apr. 2021, publisher: MDPI AG.

[61] K. Roszyk, M. R. Nowicki, and P. Skrzypczyński, "Adopting the YOLOv4 Architecture for Low-Latency Multispectral Pedestrian Detection in Autonomous Driving," *Sensors*, vol. 22, no. 3, p. 1082, Jan. 2022, publisher: MDPI. [Online]. Available: https://www.mdpi.com/1424-8220/22/3/1082

[62] H. Zhang, E. Fromont, S. Lefevre, and B. Avignon, "Guided attentive feature fusion for multispectral pedestrian detection," in *Proceedings - 2021 IEEE Winter Conference on Applications of Computer Vision, WACV 2021.* Institute of Electrical and Electronics Engineers Inc., Jan. 2021, pp. 72–80.

[63] J. Redmon, "Darknet: Open source neural networks in C," 2013. [Online]. Available: http://pjreddie.com/darknet/

[64] P. Dollár, "Piotr's Computer Vision Matlab Toolbox (PMT)." [Online]. Available: https://github.com/pdollar/toolbox

[65] C. Li, D. Song, R. Tong, and M. Tang, "Illumination-aware faster R-CNN for robust multispectral pedestrian detection," *Pattern Recognition*, vol. 85, pp. 161–171, Jan. 2019, arXiv: 1803.05347 Publisher: Elsevier Ltd.

[66] J. Liu, S. Zhang, S. Wang, and D. Metaxas, "Improved annotations of test set of kaist." [Online]. Available: http://paul.rutgers.edu/~jl1322/multispectral.htm

[67] J. Kim, H. Kim, T. Kim, N. Kim, and Y. Choi, "MLPD: Multi-Label Pedestrian Detector in Multispectral Domain," *IEEE Robotics and Automation Letters*, vol. 6, no. 4, pp. 7846–7853, Oct. 2021, publisher: Institute of Electrical and Electronics Engineers Inc.

[68] J. Deng, W. Dong, R. Socher, L.-J. Li, Kai Li, and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition.* IEEE, Jun. 2009, pp. 248–255. [Online]. Available: https://ieeexplore.ieee.org/document/5206848/

[69] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, "Microsoft COCO: Common Objects in Context," May 2014, arXiv: 1405.0312. [Online]. Available: http://arxiv.org/abs/1405.0312

[70] M. Verucchi, G. Brilli, D. Sapienza, M. Verasani, M. Arena, F. Gatti, A. Capotondi, R. Cavicchioli, M. Bertogna, and M. Solieri, "A Systematic Assessment of Embedded Neural Networks for Object Detection," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA).* IEEE, Sep. 2020, pp. 937–944. [Online]. Available: https://ieeexplore.ieee.org/document/9212130/

[71] L. Roeder, "Netron, Visualizer for neural network, deep learning, and machine learning models," Dec. 2017. [Online]. Available: https://github.com/lutzroeder/netron