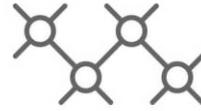




TECHNISCHE
UNIVERSITÄT
WIEN



Institut für
Computertechnik
Institute of
Computer Technology

A MASTER THESIS ON

Optimization of Siamese Object Tracking Networks

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

Diplom-Ingenieur

(Equivalent to Master of Science)

in

Embedded Systems 066 504

by

Alexander Ludwig

01525371

Supervisor(s):

Univ.Prof. Dipl.-Ing. Dr. techn. Axel Jantsch

Dipl.-Ing. Martin Lechner

Vienna, Austria

June 2023



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Visual Object Tracking (VOT) is a task where the trajectory and state of an arbitrary object shall be estimated throughout an image sequence. The only information given to the tracker is the initial ground truth location of this object. Siamese object tracking networks emerged as one main paradigm in the area of VOT and led to significant improvements. Annual held tracking challenges, like the VOT challenge, push the performance of these trackers and present advanced tracking algorithms. This thesis takes a specific tracking algorithm and presents optimization methods to run the tracker in real-time on resource-constrained hardware. It is shown that quantizing and pruning the tracking model leads to a significant increase in inference speed while maintaining the tracking quality (accuracy and robustness) is only valid for quantization. Previously, siamese object trackers were deployed on servers with high computational power. Due to the complexity of such networks, executing them on embedded devices for applications with real-time constraints was not possible. The results highlight the efficiency of quantization and pruning techniques for optimizing convolutional neural networks on embedded devices. Pruning the backbone of the Siamese object tracker enables execution up to 94 Frames Per Second (FPS) at a compression factor of 0.26 - but this comes with the cost of losing significant tracking quality and makes these trackers not suitable for real-world applications. A Siamese object tracker optimized with the NVIDIA TensorRT library is able to run at 50 FPS at 16-bit floating point precision on the NVIDIA Xavier AGX. In contrast, the non-optimized tracker runs at 11 FPS at 32-bit floating point precision on the same device with similar tracking quality. To summarize, the implemented optimizations were able to gain inference speed by factor 4.5.

Kurzfassung

Bei Visual Object Tracking (VOT) soll die aktuelle Position eines beliebig wählbaren Objekts durchgehend über eine Sequenz von Bildern berechnet werden. Dem Tracker liegt hierfür lediglich die initiale Position dieses Objekts vor. Siamese Object Tracking Networks haben sich in den letzten Jahren im Bereich des VOT etabliert und zu signifikanten Fortschritten geführt. Jährlich stattfindende Wettbewerbe wie die VOT Challenge treiben die Leistungsfähigkeit dieser Tracker voran und präsentieren fortschrittliche Tracking-Algorithmen. In dieser Arbeit wird ein spezifischer Tracking-Algorithmus betrachtet und Optimierungsmethoden vorgestellt, um den Tracker in Echtzeit auf Geräten mit beschränkter Hardwareressourcen auszuführen. Es wird gezeigt, dass die Quantisierung und das Pruning des Tracking-Modells zu einer signifikanten Steigerung der Inferenzgeschwindigkeit führen, während die Tracking-Qualität (Genauigkeit und Robustheit) nur für Quantisierung erhalten bleibt. Zuvor wurden Siamese Object Tracking Networks auf Servern mit hoher Rechenleistung eingesetzt. Aufgrund der hohen Komplexität solcher Netzwerke war es jedoch nicht möglich, sie auf Embedded Hardware für Anwendungen mit Echtzeit-Anforderungen auszuführen. Die Ergebnisse unterstreichen die Effizienz von Quantisierungs- und Pruning-Techniken zur Optimierung von Convolutional Neural Networks (CNNs) auf Embedded Hardware. Durch das Pruning des Backbones des Siamese Object Trackers kann eine Ausführung von bis zu 94 Bildern pro Sekunde (FPS) bei einer Kompressionsrate von 0.26 erreicht werden - dies geht jedoch mit dem Verlust erheblicher Tracking-Qualität einher und macht diese Tracker für den Einsatz in der realen Welt ungeeignet. Ein mit der NVIDIA TensorRT Bibliothek optimierter Siamese Object Tracker kann jedoch mit einer Genauigkeit von 16-Bit Gleitkommazahlen (floating point) auf dem NVIDIA Xavier AGX mit 50 FPS ausgeführt werden. Im Vergleich dazu läuft der nicht optimierte Tracker mit einer Genauigkeit von 32-Bit Gleitkommazahlen auf demselben Gerät mit einer Geschwindigkeit von 11 FPS bei ähnlicher Tracking-Qualität. Zusammenfassend kann man sagen, dass die implementierten Optimierungen zu einer Steigerung der Inferenzgeschwindigkeit um den Faktor 4.5 führen.

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.

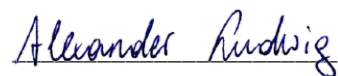
Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Copyright Statement

I, Alexander Ludwig, 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:



Vienna, Austria, June 2023

Alexander Ludwig

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

Abstract	iii
Kurzfassung	iv
1 Introduction	1
1.1 Problem definition	2
1.2 Main Contribution	3
2 Theory and State of the Art	5
2.1 Convolutional Neural Networks	5
2.1.1 Main Components	6
2.1.2 Training	10
2.1.3 Residual Networks	12
2.1.4 Object Detection	14
2.2 Visual Object Tracking	16
2.2.1 Discriminative Correlation Filters	18
2.2.2 Siamese Networks	19
2.2.3 Visual Transformers	23
2.3 Optimizing Convolutional Neural Networks For Resource Constrained Hardware	24
2.3.1 Pruning	24
2.3.2 Quantizing	27
2.3.3 Knowledge Distillation	28
3 Methodology	29
3.1 SiamRPN++ [1]	29
3.1.1 Training	31
3.2 Hardware Setup	33

3.3	Evaluation Metrics	34
3.3.1	Evaluation protocols and performance measures	34
3.3.2	Testing Dataset	37
3.3.3	Inference latency	37
3.3.4	Power consumption	38
3.4	Conversion into ONNX	38
3.5	Optimization with TensorRT	41
3.6	Pruning the Backbone	43
3.6.1	ResNet-50 of SiamRPN++	44
3.6.2	Pruning workflow	44
4	Experiments and Results	47
4.1	Baseline model	47
4.2	TensorRT engines	48
4.2.1	Mixing precisions	54
4.2.2	Pruned backbone	57
4.2.3	TensorRT engines with pruned backbone	60
4.3	GPU Power and energy consumption	62
5	Conclusion and future work	65
5.1	Conclusion	65
5.2	Future work	66
	Bibliography	67

List of Tables

4.1	Baseline model results: The first row are the results presented in the SiamRPN++ paper [1] and serve as a benchmark (except FPS).	47
4.2	Tracking results on VOT2018 for TensorRT engines	49
4.3	Tracking results when mixing the engine’s precisions.	55
4.4	Results for finetuning the pruned networks. <i>Note: The network with 45M parameters is the non-pruned network.</i>	59
4.5	Tracking results on VOT2018 for networks with original number of parameters (45.5M) and pruned ResNet50 backbones (30M, 22.5M and 12M).	60
4.6	GPU Power and energy consumption of the networks with the baseline ResNet50 backbones	62
4.7	GPU Power and energy consumption of the networks with pruned ResNet50 backbones. The percentage difference for each precision mode is related to the power consumption value of the 45.5M parameters backbone.	63

List of Figures

2.1	Fully-connected neural network with 3 input neurons, 2 output neurons and one hidden layer.	6
2.2	Calculation scheme for a single neuron.	6
2.3	CNN: connect only a small region of the input image to a neuron [2].	7
2.4	The principle of convolution: n 2×2 kernels are convolved over the n 3×3 input feature maps to obtain the n 2×2 output feature maps (Equation 2.1).	8
2.5	Activation functions: (a) Sigmoid; (b) Hyperbolic Tangent; (c) ReLU; (d) Leaky ReLU with $a = 0.1$	9
2.6	Example of 2×2 max pooling and average pooling with a stride of 2	10
2.7	Training error (left) and test error (right) on CIFAR-10 [3] with 20-layer and 56-layer “plain“ networks [4].	12
2.8	Residual block [4].	13
2.9	Residual block in bottleneck design [4].	14
2.10	Region Proposal Network (RPN) [5]	15
2.11	Tracking challenges due to real-world effects. All snapshots are from the VOT2018 challenge dataset [6]	17
2.12	Fully-convolutional Siamese architecture SiamFC [7]. Φ denotes a convolutional function.	20
2.13	Network architecture of SiamRPN [8]	21
2.14	CURL [9]: Not only channels inside the residual block (red) are pruned, but also output channels (green). In each rectangle (representing a convolutional block), the first two numbers indicate the output channels and input channels, respectively.	26
3.1	Target position probabilities when using random shift. [1]	30
3.2	Network architecture of SiamRPN++ [1]. The outputs of three Siamese RPNs, which take features from different depth levels, are fused into a output prediction.	31

3.3	Training images (left) from ImageNet DET [10] with their target (middle) and search (right) crops.	32
3.4	NVIDIA Jetson Xavier AGX	33
3.5	IoU: Illustration of notations from Equation 3.2	35
3.6	Expected average overlap on a N_S frames long sequence [11].	35
3.7	Left up: the Expected Average Overlap (EAO) curve. Left bottom: the sequence length pdf. Right: plot of the EAO [12].	36
3.8	Illustration of downsampling and cropping template features in the neck section of the network.	39
3.9	Introduced structure to separate SiamRPN++ [1] into three networks.	40
3.10	TensorRT PTQ workflow [13]	43
3.11	Architecture of the SiamRPN++ ResNet-50 [1]. The crossed-out area indicates the missing modules compared to the standard ResNet-50.	44
4.1	Comparison of quality (EAO) and speed of the baseline tracker (PyTorch FP32) and the tracker with TensorRT engines.	49
4.2	Layer-type latencies (average in ms) of the target engine for FP32, FP16 and INT8.	51
4.3	Layer-type latencies (average in ms) of the search engine for FP32, FP16 and INT8.	51
4.4	Layer-type latencies (average in ms) of the xcorr engine for FP32, FP16 and INT8.	52
4.5	Evaluation with respect to visual attributes on the VOT2018 testing dataset.	54
4.6	FPS results for the tracker with search engine in FP32, FP16 and INT8.	56
4.7	EAO results for the tracker with xcorr engine in FP32, FP16 and INT8.	57
4.8	Top-1 accuracy for training the backbone on the ImageNet classification dataset with the two attached layers (details in section 3.6.2)	58
4.9	Visualization of the finetuning of the pruned networks from table 4.4	59
4.10	Comparison of quality (EAO) and speed of trackers with the original size backbone (45, 5M parameters) and pruned backbones (30M, 22.5M and 12M).	62
4.11	Visualization of the GPU power consumption for the networks with pruned ResNet50 backbones.	64

Acronyms

CNN Convolutional Neural Network.

DCF Discriminative Correlation Filter.

EAO Expected Average Overlap.

FPGA Field Programmable Gate Array.

FPS Frames Per Second.

GPU Graphics Processing Unit.

HMI Humane Machine Interaction.

IoU Intersection over Union.

ONNX Open Neural Network Exchange.

PTQ Post Training Quantization.

ReLU Rectified Linear Unit.

ResNet Residual Neural Network.

RoI Region of Interest.

RPN Region Proposal Network.

SGD Stochastic Gradient Descent.

SN Siamese Network.

SoC System on Chip.

ViT Vision Transformer.

VOT Visual Object Tracking.

Chapter 1

Introduction

Visual Object Tracking (VOT) is the task of estimating the location of an object of interest when only the initial ground truth position of the object is known. The main difficulty is handling appearance changes (such as deformation and occlusion) and similar background objects while maintaining accurate bounding box estimation. Visual object trackers have widespread application fields, for example, autonomous driving and visual surveillance. In the past, visual object tracking algorithms heavily relied on features such as color histograms, texture descriptors, and motion information [14]. These features were used to represent the object being tracked and compare it to potential regions of interest in subsequent frames. However, these traditional methods struggled with complex scenarios involving substantial changes in object appearance, occlusion, or motion blur.

Convolutional Neural Networks (CNNs) have revolutionized the application field of computer vision. They show remarkable performance in tasks such as image classification and object detection. The victory of AlexNet [15] in the ImageNet classification challenge 2012 [10] marked a significant point, igniting widespread attention of deep CNNs in both research and industry domains. CNNs have become the foundation of modern computer vision systems. Inspired by the structure and functionality of the human brain's visual cortex, they can learn and extract visual patterns, from simple edges and textures to more complex shapes.

Siamese object tracking utilizes a one-shot learning method, where certain properties about a category of objects are learned using a single sample. With the help of Siamese Networks (SNs) and deep Convolutional Neural Networks (CNNs), massive improvements have been achieved in the application field of Visual Object Tracking (VOT). Siamese object tracking networks have emerged as an important technique in computer vision applications, allowing for robust and accurate tracking of generic objects across consecutive frames. These networks leverage deep CNNs to learn discriminative features and match them across consecutive frames of videos. While the accuracy of Siamese object tracking

networks has been widely acknowledged, their deployment on embedded hardware presents several computational challenges.

Embedded hardware (or devices) refers to computer systems that are specialized for a specific type of task, often with additional constraints on size, power consumption, and cost. These systems are typically embedded in a larger system. For example, a System on Chip (SoC) or Field Programmable Gate Array (FPGA) operating as an assistance system inside an automotive system. Due to all the constraints, embedded devices have limited computational power compared to general-purpose computing systems or servers. This requires the development of optimized software that is tailored to the embedded hardware and the function to be performed.

NVIDIA offers the Jetson hardware platform, which is specially designed for accelerating machine learning algorithms on embedded hardware. These devices leverage the CUDA parallel computing architecture, which allows utilizing the Graphics Processing Unit (GPU) for machine learning processing. This allows the inference of CNNs on the embedded devices itself, vanishing the need for cloud-based processing or external servers. All of the computation can be done locally on the device, enabling real-time inference and decision-making capabilities.

This thesis addresses the following research question: Which impact do different CNNs optimization techniques have on a Siamese object tracking network? Therefore a method for optimizing the Siamese object tracker proposed by Li et al. [1] with the NVIDIA TensorRT library is described. The goal is to execute this tracker at real-time speed (30 Frames Per Second (FPS)) while maintaining tracking quality (Expected Average Overlap (EAO), accuracy, and robustness). Additionally, the impact of pruning the tracker backbone is investigated regarding tracking quality. Finally, the effect of the optimizations on the power consumption of the NVIDIA Xavier AGX is measured.

1.1 Problem definition

Deploying Siamese object tracking networks on resource-constrained embedded systems presents significant challenges due to limited computational resources, memory constraints, and energy efficiency requirements. These networks achieve first-class tracking results, but due to their complexity, they are only suitable for real-time applications on resource-constrained hardware to a limited extent. Additionally, research focuses on improving the tracking quality and not executing such deep CNNs networks on embedded devices like the NVIDIA Xavier AGX. Therefore the objective of this work is to use a state-of-the-art Siamese object tracking network (SiamRPN++ [1]), optimize it for an embedded hardware platform (NVIDIA Xavier AGX), and analyze the impact of different CNN optimization techniques.

The main steps of this work are the following:

- Evaluate the performance of the SiamRPN++ tracker on the NVIDIA Xavier AGX.
- Convert the SiamRPN++ PyTorch model to the ONNX format, optimize with the NVIDIA TensorRT library, and evaluate the impact on execution speed and tracking quality.
- Prune the SiamRPN++ ResNet-50 backbone and evaluate the impact on execution speed and tracking quality.

1.2 Main Contribution

The main contributions of this work are the following:

- Deploying a siamese object tracking network (SiamRPN++ [1]) on the NVIDIA Xavier AGX and optimize it with the NVIDIA TensorRT library.
- Implement a pruning method for the backbone of the tracking network.
- Benchmark and analyze the impact of these optimization techniques.

Chapter 2

Theory and State of the Art

2.1 Convolutional Neural Networks

Nowadays Convolutional Neural Networks (CNNs) receive a lot of attention both in research and in industry. They have widespread applications and are most commonly used for Image Processing and Computer Vision. Inspired by the natural visual perception of living creatures, Fukushima et al. [16] first proposed the Neocognitron architecture in 1980 which can be considered as the predecessor of CNNs.

The origin of modern CNNs comes with the introduction of LeNet by LeCun et al [17]. At the time LeCun et al. wrote their paper (1998), machine learning techniques were based on hand-designed heuristics. With the task of recognizing handwritten digits, they showed that hand-crafted feature extraction could be replaced by carefully designed learning machines that operate directly on pixel images. LeNet has a multi-layer structure, which is a characteristic for CNNs and can be trained with the backpropagation algorithm [18]. But due to limited training data and computation resources at that time, the network lacks in performing on complex problems like large-scale image classification [19].

The ImageNet classification challenge named “ImageNet large scale visual recognition challenge (ILSVRC)” [10] is mainly responsible for subsequent developments in the area of CNNs. In 2012 Krizhevsky et al. proposed a CNN architecture most popular known as AlexNet [15] that showed significant improvements in the image classification task compared to previous architectures. The structure of AlexNet is similar to LeNet, but AlexNet has a higher number of layers and therefore can be considered more deep. Since then the trend was to make networks deeper to get better feature representations. Making a CNN deeper provides the ability to learn complex representations at different levels of abstraction [4]. However, increasing the depth of a network also increases complexity and difficulty to optimize.

Below the main components of CNNs are introduced, followed by training and optimization techniques. Then CNNs for object detection, the concept of residual networks is presented, and finally, the concept of residual networks is discussed.

2.1.1 Main Components

Regular neural networks transform the incoming input vector through hidden layers. The neurons of one layer are connected to all neurons of the previous layer (fully-connected, Figure 2.1). In the neural network from figure 2.1, each circle represents a neuron and the arrows represent the weights. Neurons receive i inputs denoted as x_i , which are multiplied by i weights denoted as w_i (Figure 2.2). The products of the inputs and weights are then summed up, forming the input for the activation function σ . The output of the activation function is the output of the neuron.

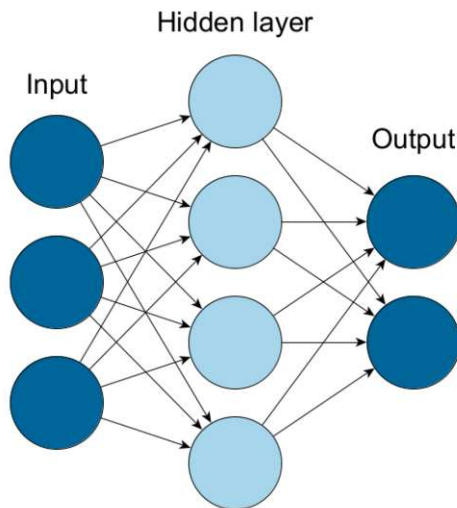


Figure 2.1: Fully-connected neural network with 3 input neurons, 2 output neurons and one hidden layer.

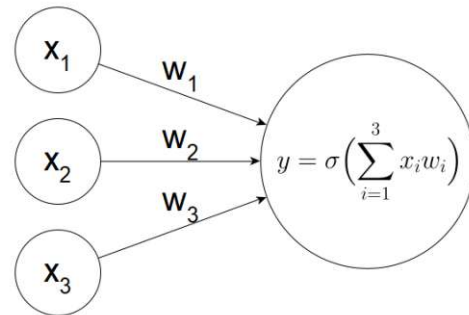


Figure 2.2: Calculation scheme for a single neuron.

Classical neural networks are not suited for handling image inputs that can contain thousands of pixels and up to three color channels. For an RGB image with dimensions of $32 \times 32 \times 3$ (width \times height \times depth), a single neuron would require $32 \times 32 \times 3 = 3072$ connections (from each input pixel to the neuron). Since a single neuron will not be enough for any useful image processing, more neurons are required. But when using more neurons in a fully-connected fashion, the number of connections and network parameters required becomes drastically large.

A more efficient method is to connect only a small region of the input image spatially to a neuron, but to the full depth (Figure 2.3). For example, a single neuron can be connected only to $3 \times 3 \times 3$ neurons from the input image. An additional simplification is to keep these $3 \times 3 \times 3$ weights fixed for

all neurons of the next layer. Therefore only 27 weights are needed to connect $32 \times 32 \times 3$ neurons to 32×32 neurons. Fixing the weights is similar to sliding a $3 \times 3 \times 3$ window spatially over the input neurons - also called convolution. This characteristic is what gives Convolutional Neural Networks (CNNs) their name.

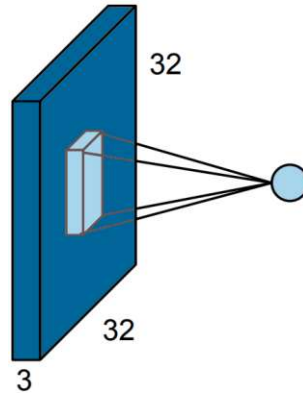


Figure 2.3: CNN: connect only a small region of the input image to a neuron [2].

Although many different CNN architectures exist in the literature, they do not differ in their core components. Convolution layers are responsible for extracting useful features and their output is fed into the activation functions, which add non-linearity in feature space. Non-linearity helps in learning semantic differences in images [20]. Without non-linearity, neural networks would reduce to a single linear layer, which is unable to handle tasks like object detection. Pooling reduces the spatial size of the feature maps and is useful for extracting dominant features. The fully-connected layer is often referred as the “output layer“ that computes the resulting class scores (for object classification). In recent CNN architectures like ResNet [4] fully-connected are replaced by convolutional layers [21].

Convolutional Layers

The convolution layer is the heart of a CNN. It produces output feature maps by convolving filter kernels across the width and height of input feature maps (Figure 2.4). The learnable parameters of a convolution layer are the n $K \times K$ filter kernels. These filter kernels are moved across each input feature map and at each position, the sum of the element-wise product between the kernel values and feature map values is calculated (Equation 2.1). The convolution generates a 3D output feature map by piling n 2D output feature maps produced by each filter kernel. The resulting size of the output feature map is given by $(W - F + 2P)/S + 1$. The parameter stride S controls the amount of filter kernels of size F moving over the input feature maps of size W . With a stride of 1, the kernel is moved one pixel at a time. Another parameter is padding P , which adds P extra pixels around the boundary of the input feature map to prevent losing pixels in the output feature map. In other words, padding is

used to control the spatial size of the output feature maps. The common value of the added pixels is zero (zero-padding), but also the average of the surrounding region of the border pixel is used.

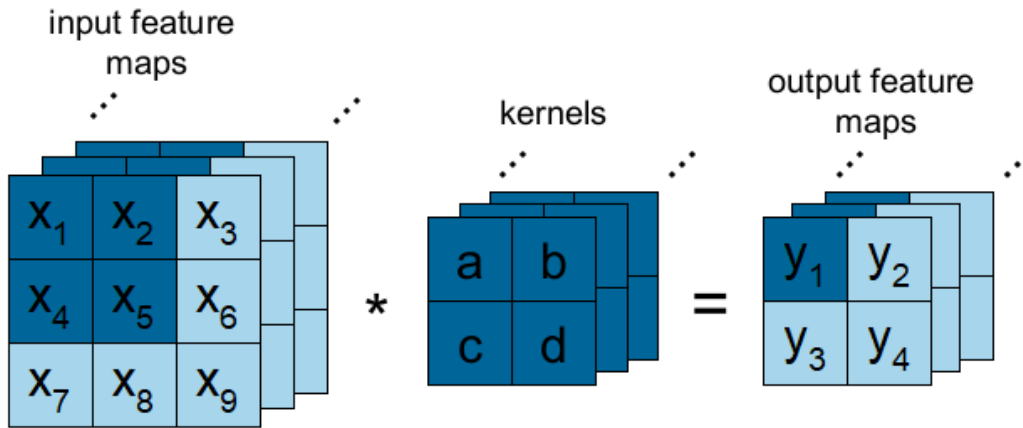


Figure 2.4: The principle of convolution: n 2×2 kernels are convolved over the n 3×3 input feature maps to obtain the n 2×2 output feature maps (Equation 2.1).

$$\begin{aligned}
 y_1 &= x_1 * a + x_2 * b + x_4 * c + x_5 * d \\
 y_2 &= x_2 * a + x_3 * b + x_5 * c + x_6 * d \\
 y_3 &= x_4 * a + x_5 * b + x_7 * c + x_8 * d \\
 y_4 &= x_5 * a + x_6 * b + x_8 * c + x_9 * d
 \end{aligned}
 \tag{2.1}$$

Activation Functions

Convolutional layers are typically followed by an activation function, in order to add non-linearity to the network. Without any activation function there would be a linear relationship between the input and output of a network and the network would not be able to learn complex functions. CNNs, and in general neural networks are required to learn any complex function provided to them [22].

The activation function is an element-wise operation on each input feature map and therefore the dimension of the output feature maps does not change. Activation functions are typically denoted as *ReLU* because the Rectified Linear Unit (ReLU) is the most commonly used in CNNs. Previously the sigmoid function and the hyperbolic tangent function were common choices.

The sigmoid function (Figure 2.5a) takes real input values and maps them into a range between 0 and 1. Nowadays it is no longer used anymore due to two main problems: first, when activation values saturate around 0 and 1, the gradient vanishes. Second, the activation values are not zero-centered. The hyperbolic tangent function (Figure 2.5b) is zero-centered but also saturates (around -1 and 1). In

Rectified Linear Unit (ReLU) (Figure 2.5c), the activation is thresholded at zero which greatly reduces the computation costs and accelerates training convergence compared to sigmoid and hyperbolic tangent functions. Leaky ReLU (Figure 2.5d) is an improved version of ReLU that tries to fix the issue of “dying ReLU” by having a positive slope for $x < 0$ instead of being zero. For more details regarding dying ReLU and activation functions in general, refer to [23].

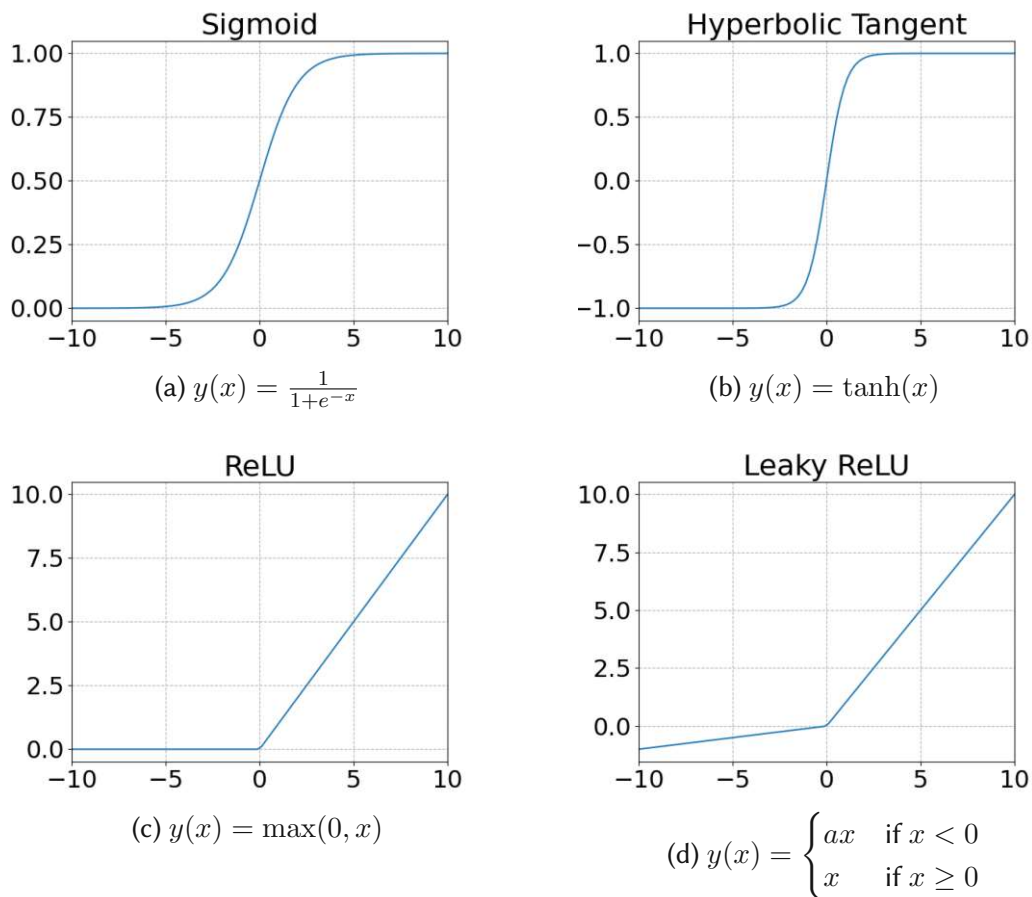


Figure 2.5: Activation functions: (a) Sigmoid; (b) Hyperbolic Tangent; (c) ReLU; (d) Leaky ReLU with $a = 0.1$

Pooling Layers

Pooling layers are usually inserted periodically between convolution layers and contribute to parameter reduction by reducing the spatial size of the feature maps. The common way of pooling is applying a 2×2 filter, that performs a max operation, on every depth slice of a feature map (Figure 2.6). Another popular pooling method is average pooling, where a 2×2 filter computing the average is applied on the feature map. Pooling has two hyperparameters: the spatial extent of the pooling filter (most common 2×2) and the stride.

These pooling methods operate locally across a certain input feature map. In contrast, global pool-

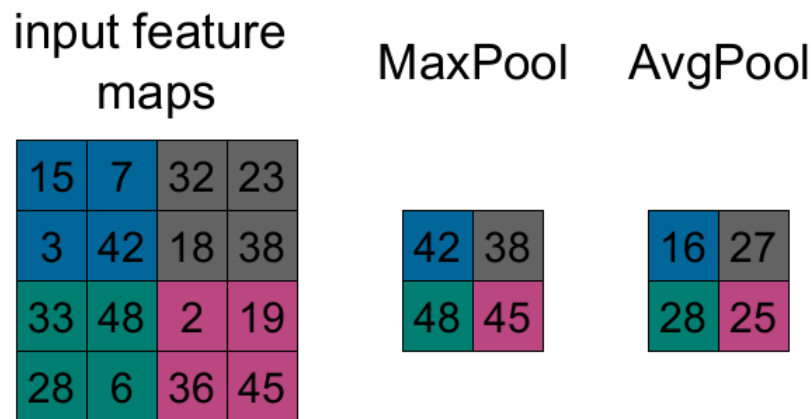


Figure 2.6: Example of 2×2 max pooling and average pooling with a stride of 2

ing operates globally across all spatial dimensions of the input feature map. Global pooling is commonly used in the final layers of a network for global feature aggregation and classification purposes.

An alternative approach for pooling is using a convolutional layer with a stride greater than 1. This also leads to non-overlapping receptive fields, resulting in a downsampling effect. As a result, the spatial dimensions of the feature maps are reduced, achieving a similar outcome to pooling.

Normalization Layers

Training deep CNNs requires a careful selection of hyperparameters (especially learning rate and initial parameters) as small changes to the network parameters amplify as the network gets deeper. Ioffe et al. [24] found that change in the distribution of internal nodes of a deep network slows down the training convergence. To overcome this problem, they propose batch normalization to reduce the internal covariate shift. This normalization step fixes the means and variances of layer inputs and is able to reduce the dependence of gradients on the scale of the parameters. Santurkar et al. [25] claim in their paper that the reduction of the internal covariate shift is not the reason for the success of batch normalization, but that batch normalization makes the optimization landscape significantly smoother. However, the positive effect of batch normalization on training deep CNNs is evident and used in popular networks like Residual Neural Networks (ResNets) [4].

2.1.2 Training

The process in which the previously mentioned parameters of CNN layers are determined is called training. In the beginning, the parameters of a CNN have to be initialized. Initialization itself is crucial for training convergence. Arbitrary initialization can slow down or completely stall the convergence process, so He et al. [26] introduced He initialization, which is a follow-up paper of the Xavier initial-

ization from Glorot and Bengio [27]. Glorot and Bengio showed that $\sigma^2 = 1/N$ is the optimal value when initializing a neural network with samples from a normal distribution $\mathcal{N} = (\mu, \sigma^2)$. Because according to He et al., Xavier initialization does not work well for neural networks with ReLU activation functions, they proposed He initialization with $\sigma^2 = 2/N$. Kumar [28] gives detailed insights regarding the initialization of deep neural networks.

During training, images from large datasets (e.g. ImageNet [10]) are loaded into the neural network in a so called forward pass. During the forward pass, the input data is pushed through the CNN to generate output predictions. This output is then compared against the expected output using a loss function. Loss functions can be divided into two major categories: regression and classification loss functions. Regression loss functions are used in regression neural networks, where the network generates continuous values (e.g. bounding box coordinates). A popular regression loss function is the Mean Squared Error (MSE) loss function (Equation 2.2), which calculates the average of the squared differences between predicted y_i and expected output \hat{y}_i . Classification loss functions are used in classification neural networks, where the network computes the category with the highest probability. When the number of categories is equal to two, Binary Cross-Entropy (Equation 2.2) loss is a common choice. It measures the difference between predicted and expected output. p_i in equation 2.2 refers to the probability of class 1, and $(1 - p_i)$ to the probability of class 0. In case the number of categories is greater than two, Categorical Cross-Entropy Loss is often used.

$$L_i = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad \text{MSE} \quad (2.2)$$

$$L_i = \frac{1}{n} \sum_{i=1}^n -(y_i * \log(p_i) + (1 - y_i) * \log(1 - p_i)) \quad \text{Binary Cross-Entropy Loss}$$

Once the loss function is computed, the network is updated in a process called backpropagation [18]. During backpropagation, the gradients of this loss function are computed with respect to the parameters (e.g. weights and biases) of the CNN. This is done using the chain rule to propagate the gradients backward through the CNN. The overall goal is to minimize the loss between predicted and expected output. The parameters of the CNN are adjusted in a way that the predicted output is closer to the expected output. In order to find the minimum of the loss function, different optimization algorithms can be used. A common optimization algorithm is stochastic gradient descent (SGD). SGD iteratively updates the parameters of the network based on the gradients of the loss function. The term "stochastic" refers to the introduced randomness of input batches, which helps escape local minima of the loss

function. The learning rate defines the amount of adjustment applied to the network parameters. Usually, learning rate schedulers are used to decrease the learning rate as the training progresses.

The process of passing all images from the training dataset through the network is called one epoch. The dataset is not loaded at once, but instead divided into batches of much smaller size - this size is called batch size. The duration of the training can be selected based on various parameters: convergence of validation accuracy, convergence of the loss, or a fixed number of epochs. When the training has finished, the parameter of the CNN are frozen and not going to be changed anymore.

In practice, it is common to re-use already pretrained CNNs as training CNNs from scratch is time and resource consuming. Furthermore, training networks requires detailed knowledge about the network itself and the choice of hyperparameters (e.g. loss function, learning rate, epochs, batch size) often turns out to be tricky. So Transfer Learning is a technique, where a model is trained and developed for one task and then re-used on a second related task [29]. Usually, the dataset for the new task is significantly smaller than the original dataset.

2.1.3 Residual Networks

AlexNet [15] was the first deep CNN to be able to outperform previous approaches that were focused on hand-crafted feature learning. It showed that computer vision tasks greatly benefit from deep CNNs. But the expectation that stacking more layers would increase the network's ability to learn complex vision tasks was dampened by a degradation problem. The training accuracy gets saturated when increasing depth, and at a certain point degrades rapidly. He et al. [4] experimentally showed that the degradation of training accuracy is not caused by overfitting, instead more layers in a deep network lead to higher training error. Figure 2.7 shows that training and test error of the 56-layer network are higher than the 20-layer network.

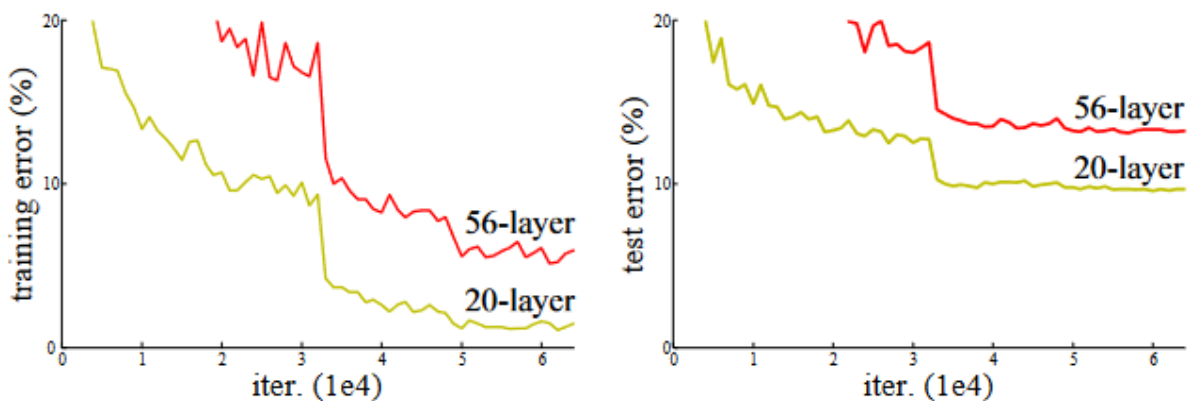


Figure 2.7: Training error (left) and test error (right) on CIFAR-10 [3] with 20-layer and 56-layer “plain” networks [4].

He et al. address this problem by introducing a new neural network layer: the residual block (Figure 2.8). Instead of fitting the stacked layers to a desired underlying mapping $\mathcal{H}(x)$, the layers are fitted to a residual mapping $\mathcal{F}(x) := \mathcal{H}(x) - x$. This results in $\mathcal{H}(x) = \mathcal{F}(x) + x$, which is realized with shortcut connections (or skip connections). The input of the residual block is bypassed along the stacked layers and added to the output of the stacked layers (assuming that input and output have the same dimensions). Shortcut connections require no additional parameters or add computational complexity. If the added layers tend to be not useful, regularization will skip over them and the overall network performance will not be affected. If the added layers tend to be useful, the overall network performance could increase.

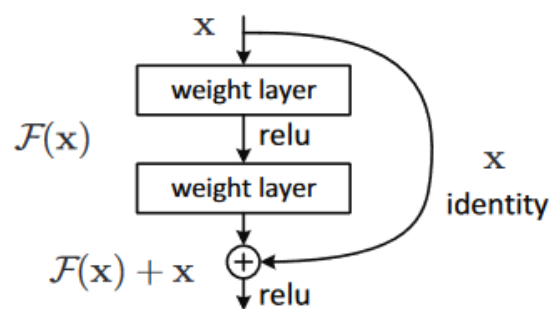


Figure 2.8: Residual block [4].

Formally, a residual block from Figure 2.8 with fully-connected layers can be denoted as

$$y = W_2\sigma(W_1x) + x \quad (2.3)$$

where x and y are the input and output vectors, W_1 and W_2 are the layer weights and σ is the ReLU function. The residual mapping to be learned can then be denoted as $\mathcal{F}(x) = W_2\sigma(W_1x)$.

If the dimensions of x and \mathcal{F} are not the same, a linear projection W_s is performed on the input vector x :

$$y = W_2\sigma(W_1x) + W_sx \quad (2.4)$$

$\mathcal{F}(x)$ is not restricted to fully-connected layers, instead, it is flexible and also applicable to convolutional layers. For convolutional layers, the addition is performed element-wise, channel by channel.

He et al. evaluate their proposed method on the ImageNet 2012 classification dataset [15] that consists of 1000 classes. Compared to its plain counterpart, a 34 layer ResNet reduces the top-1 error by 3.5% and shows considerably lower training error than a 18 layer ResNet. The second result indicates that the degradation problem is well addressed and accuracy is gained by increasing depth.

Motivated by these results, even deeper ResNets are investigated. Due to practical considerations regarding training time, the core residual block from Figure 2.8 is modified to a bottleneck design (Figure

2.9). Here a residual block consists of three convolutional layers: two 1×1 and one 3×3 convolution. The 1×1 convolutions are for reducing and then increasing (restoring) dimensions (except depth), allowing the computationally expensive 3×3 convolution to have smaller input channels.

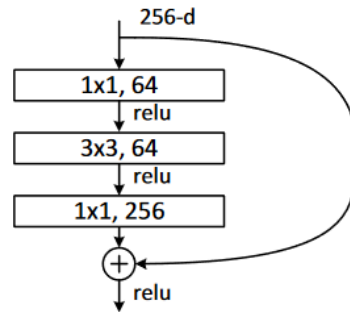


Figure 2.9: Residual block in bottleneck design [4].

In bottleneck ResNets the shortcut connections are implemented as identity shortcuts because linear projections would increase time complexity and network parameters. Replacing every 2 layer block in the 34 layer ResNet results in the popular ResNet-50. ResNet-50 has a significantly higher accuracy and the benefits of depth are greatly exhibited. The original paper [4] discusses the results and analysis in more detail.

2.1.4 Object Detection

Object detection is one of three main tasks in computer vision, where the goal is to detect every object and its associated class in the image. It is a combination of the other two tasks: image classification and object localization. Image classification is about predicting the class of an object in an image while object localization is locating these objects and indicating their location in the image with a bounding box.

Bounding boxes are rectangles determined by 4 x and y image coordinates in different formats. Common formats are (x_0, y_0, w, h) and (x_0, y_0, x_1, y_1) , with (x_0, y_0) representing the upper left corner, (x_1, y_1) the lower right corner and (w, h) the width and height of the bounding box.

The basic technique for object detection is the sliding window method, where a window of size $M \times N$ is used to search objects over the image. The actual detection is done by using a classifier over the search window to distinguish learned objects of interest from irregular objects of different classes [30]. Methods for object detection can be divided into region proposal methods (RPN) and classification-based methods. R-CNN [31], Fast R-CNN [32] and Faster R-CNN [5] are examples for Region Proposal Network (RPN), and single shot detector (SSD) [33] and You Only Look Once (YOLO) [34] are examples for classification-based methods.

Fast R-CNN fixes these drawbacks by improving speed (train and test times) and accuracy. The deep

CNN takes two inputs simultaneously: the image and multiple Region of Interests (RoIs). It produces a feature map of the input image and then for each RoI a fixed-length feature vector out of this feature map (using the RoI pooling layer and fully connected layers). For each RoI the network delivers two outputs: softmax probabilities over the object classes and per-class bounding-box regression offsets.

Fast R-CNN could improve speed and accuracy, but according to Ren et al. [5] it still has disadvantages and room for improvement. Approaches like R-CNN and Fast R-CNN do not have any method for choosing RoIs. Instead they take all (e.g. 2000) RoIs to compute the object proposal. To prevent that every RoI has to be processed, Ren et al. introduce a Region Proposal Network (RPN) that delivers the RoI proposals. Faster R-CNN consists of two modules: the fully convolutional RPN and the detector from Fast R-CNN that uses the proposed regions.

A RPN is a CNN that shares convolutional layers with the detection network, resulting in significantly lower costs for computing region proposals. First, the input image is passed through a CNN backbone network (e.g. ResNet) to compute an output feature map (like in Fast R-CNN). Then a sliding window is applied on this feature map, and multiple region proposals are predicted simultaneously at every sliding window location. The maximal number of possible region proposals at each sliding window location as k (Figure 2.10). Each sliding window is mapped to a lower dimensional feature vector (intermediate layer), which is input for the box-classification layer (*cls* layer) and the box-regression layer (*reg* layer). The *cls* layer outputs $2k$ scores for each sliding window location and the *reg* layer outputs $4k$ values encoding the coordinates of the bounding box. The scores and values are parameterized relative to k anchor boxes. The anchor boxes are centered at the sliding window and have different scale and aspect ratios (Figure 2.10). The $2k$ category scores (*cls* scores) and the $4k$ bounding box offsets are

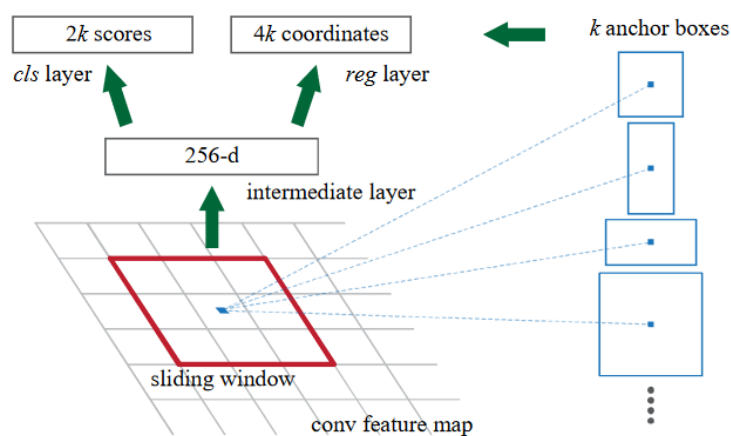


Figure 2.10: Region Proposal Network (RPN) [5]

predicted with a 1×1 convolutional network. When training the network, anchor boxes that overlap most with the ground truth bounding box are sampled and denoted as positive or activated anchor boxes. Additionally, anchor boxes with an overlap (Intersection over Union (IoU)) greater than 0.7 with

any ground truth bounding box are considered positive. Negative anchor boxes are sampled when the overlap with the ground truth bounding box is little (below an IoU of 0.3) to none. The positive anchor boxes are assigned with the object's corresponding category, while the negatives are assigned with the class background.

The *cls* layer learns to classify anchor boxes using binary cross-entropy loss. Because the positive anchor boxes may not align with the ground truth bounding box, the *reg* layer is learned to predict the offsets to the ground truth bounding box. L2 regression loss is used to learn the offsets. The $4k$ offsets are used to compute the region proposals out of the k anchor boxes.

The second module (Fast R-CNN detector) takes the region proposals as input and learns to predict the objects category inside the RoI. Because the RoIs can have different sizes, RoI is used to resize them before passing through the network.

For inference, only the top-ranked RoIs with a high classification score are used as input for the second module. The detector predicts the final categories and bounding box offsets. Duplicate bounding boxes are removed in a post-processing step named non-max suppression. Finally bounding boxes can be calculated and drawn using the offsets predicted by the detector.

Since the introduction of Faster R-CNN more modern methods like YOLO [34] or EfficientDet [35] have been developed and proposed. However, Faster R-CNN provided the basis for most of the recent developments in object detection.

2.2 Visual Object Tracking

The core challenge of VOT is to predict the state of a given object while only the initial state of this object is given. This state is often described with a bounding box, but recently also segmentation is used. The object to be tracked is not known a priori and is not constrained to any class. The tracking task consists of a classification and a regression problem. The classification problem is to distinguish the target object from the background, while the regression problem is to estimate an accurate bounding box. When learning this model, the tracker has to deal with various effects (Fig. 2.11):

- Occlusion: The target object can be partially or fully covered by another object.
- Deformation or scale variation
- Rotation of the target object or movement of the camera
- Similar objects: objects with similar appearance in the background that can easily be confused with the target itself
- Environmental factors like illumination changes and motion blur



(a) Occlusion



(b) Rotation and scale variation



(c) Illumination changes



(d) Similar objects in the background

Figure 2.11: Tracking challenges due to real-world effects. All snapshots are from the VOT2018 challenge dataset [6]

The goal is to follow the object's trajectory despite these visual distractions and variations in the visual appearance. Pflugfelder [36] described the task of tracking in computer vision as the stochastic problem of estimating and predicting random variables such as object appearance, position, dynamics and behavior. In contrast to other inference problems, visual tracking is considered probabilistic as many factors of the problem are typically unknown or uncertain.

The process of visual object tracking typically involves several steps. The first step is to initialize the tracker with the object of interest in the initial frame of the video sequence. This can be done by

providing the coordinates of the initial bounding box. In the next step, the tracker computes a representation of the object which captures the visual appearance. This allows comparison of the object in subsequent frames. Common representations include histograms of color, texture and shape features, as well as features extracted from deep neural networks.

Next, the actual core operation of visual object tracking is performed. The object's position and scale are estimated in each frame of the video sequence based on the previously generated features. This can be done using tracking algorithms such as correlation filters [37–41], Kalman filters [42, 43] or deep neural networks.

Visual object trackers can handle the loss of the object of interest differently. Trackers without implemented re-detection have to be initialized again and can not explicitly detect occlusion (“Short-term trackers”). “Long-term trackers“ detect tracking failures and implement explicit target re-detection [44]. In order to reflect changes in the object's appearance, the object representation may need to be updated as new frames become available. This can be done by using update mechanisms, such as online learning or adaptive appearance models (cites).

In recent years VOT has drawn increasing research attention and has widespread application domains which include [45]:

- Autonomous driving: e.g. assistance systems that track other vehicles on the road, autopilot in cars
- Humane Machine Interaction (HMI): e.g. eye gaze tracking for disabled people
- Visual surveillance and security systems: e.g. monitoring human activities
- Traffic Monitoring: e.g. monitoring traffic flow or detecting traffic accidents
- Medical diagnosis and imaging: e.g. tracking of the ventricular wall

Due to the lack of established methodologies for objective comparison, Kristan et al. started the Visual Object Tracking Challenge ¹ in 2013 [46]. Since then, the challenge has been held annually and provides a common platform for discussing and evaluating advancements in the field of visual tracking.

Over the years three design approaches emerged as the most promising ones in VOT: Discriminative Correlation Filters (DCF), Siamese Networks (SNs) and recently transformer-based trackers [47].

2.2.1 Discriminative Correlation Filters

DCF train correlation filters online on the region of interest by minimizing a least squares loss. These correlation filters are convolved over a search region and the location with the maximum value of this convolution is the new target location. The DCF algorithm is based on the observation that image

¹<https://www.votchallenge.net/>

intensity values within a small patch are highly correlated, and this correlation can be used to track objects of interest.

Tracking with DCFs involves several steps. First, features have to be extracted from the target object and the search window in the next frame. Then, correlation filters, which can maximize the correlation between features from the target object and the search window, have to be learned. The learned filters are then applied to the search window of the next frame to estimate the correlation with the target object. The position of the maximum correlation indicates the position of the target object. Finally, the filters get updated so that changes in the object's appearance and scale can be incorporated.

Bolme et al. [39] produce convolution filters by the minimum output sum of squared error (MOSSE) method. These filters are more robust to appearance changes and are better at discriminating targets from the background. The correlation is computed in the Fourier domain using Fast Fourier Transform (FFT). This enables creating a fast tracker, however, the bottleneck of this tracker is the computation of the forward and inverse FFTs. The online update of filters is done by putting more weight on recent frames and decaying the effect of previous frames exponentially over time.

2.2.2 Siamese Networks

Using deep CNNs for computer vision problems has become popular, but for tracking objects in real-time video, not enough labeled data has been available, which makes it difficult to use deep learning in this context. To address the problem of limited labeled data, pre-trained deep CNNs originally trained for different tasks have been tried: using the network's internal representation as features for "shallow" methods like correlation filters, and fine-tuning multiple layers of the network using Stochastic Gradient Descent (SGD). While the latter approach achieves state-of-the-art results, it is not real-time applicable. And the former approach does not utilize end-to-end learning to its fullest potential.

Held et al. [48] showed that it is possible to train a tracker offline using videos of objects and execute tracking in real time. Their tracking network GOTURN is trained completely offline, so at test time the weights are frozen and no online update is necessary. This offline training enables the tracker to track novel objects in a fast, robust and accurate manner. Held et al. combined offline training with a one-pass regression (for computing the location of the object) to generate a significant speed-up compared to previous approaches.

Bertinetto et al. [7] state that the method from Held et al. does not possess intrinsic invariance to translation of the next frame. Therefore GOTURN has to be trained with examples in all positions using dataset augmentation. Competitive methods (like GOTURN) at that time trained on video sequences belonging to the same domain used by tracking benchmarks. Bertinetto et al. showed that a CNN can be trained without videos from the same domain as the testing dataset.

Siamese Object Tracking Networks first have been introduced by Bertinetto et al. [7] as a way to improve tracking performance. Until then, the most successful trackers learned a model of the object's appearance in an online fashion, requiring significant computational resources. They formulated a function $f(z, x)$, which evaluates an exemplar image z against a candidate image x of equal size and returns a high score if the two images correspond to the same object and a low score when they do not. To locate an object in a new image, all conceivable positions are selected and the candidate with the highest similarity to the object's prior appearance is selected. For their experiments the initial appearance of the object is used as the exemplar. The function f is learned by utilizing a set of videos with annotated object trajectories.

The proposed architecture (Figure 2.12) is fully convolutional with respect to the candidate image x . A significant benefit of utilizing a fully-convolutional network is that much larger sizes of the search image can be used. And the network will compute the similarity across all translated sub-windows on a dense grid in a single evaluation.

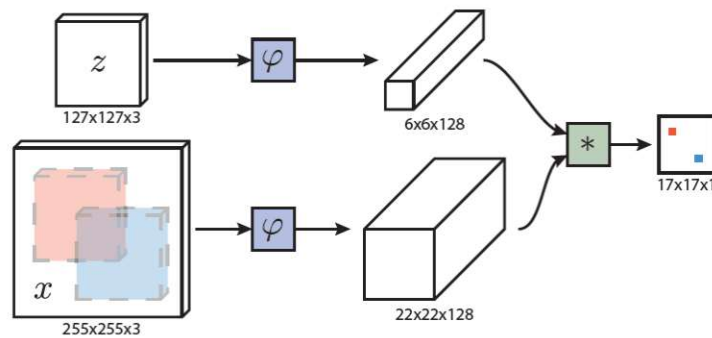


Figure 2.12: Fully-convolutional Siamese architecture SiamFC [7]. Φ denotes a convolutional function.

The feature maps after applying Φ are then cross-correlated:

$$f(z, x) = \varphi(z) * \varphi(x) + b\mathbb{1} \quad (2.5)$$

where b denotes a signal which takes value $b \in \mathbb{R}$ in every location. The result is a score map on a finite grid. While tracking, the search image is centered on the previous target position. By determining the maximum score's position in relation to the score map's center and then multiplying it by the network's stride, the target's displacement from frame to frame is computed.

Training images are organized in exemplar-candidate pairs centered on the target position, and generated from a dataset of annotated videos. The class of the object is ignored during training. For more details regarding training, implementation and evaluation, refer to the paper [7].

The previously described SiamFC tracker also has a major drawback: although it is operating in real-time, the accuracy and robustness are unsatisfying when compared to SOTA DCFs approaches. Bertinetto et al. already proposed a possible method for refinement: bounding box regression. Li et al. [8] therefore present a Siamese Region Proposal Network (RPN) inspired by the work of Ren et al. “Faster-RCNN” [5].

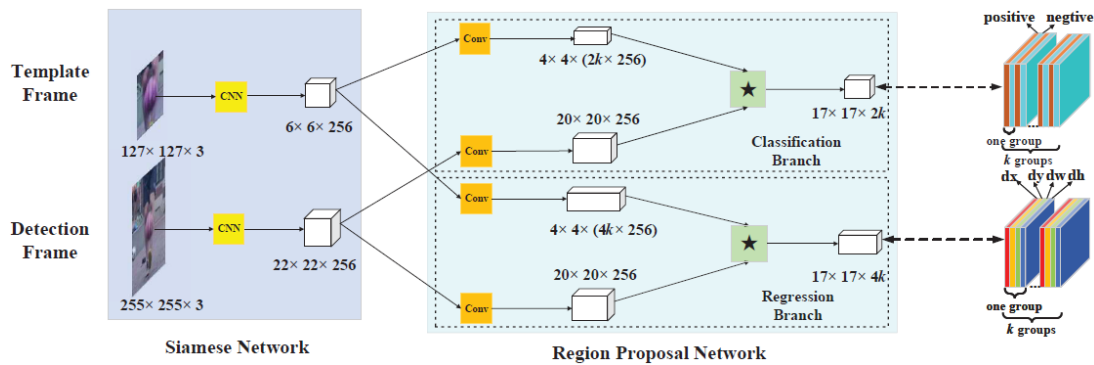


Figure 2.13: Network architecture of SiamRPN [8]

The network architecture of SiamRPN depicted in Figure 2.13 is divided into two subnetworks: the SN with the feature extractor (Li et al. use a modified AlexNet [15]) and the RPN. Like in SiamFC, the feature extractor consists of two branches that share the same CNN weights: the template branch and the detection branch. The template branch takes the image z with the object to track as input, while the detection branch takes the current frame x of the video as input. The RPN also consists of two branches: the foreground-background classification branch and the proposal regression branch. Every output from the SN is split into two branches: $[\varphi(z)_{cls}, \varphi(z)_{reg}]$ and $[\varphi(x)_{cls}, \varphi(x)_{reg}]$. $\varphi(z)_{cls}$ and $\varphi(z)_{reg}$ serve as the convolution kernels for the following convolutions:

$$A_{w \times h \times 2k}^{cls} = \varphi(x)_{cls} * \varphi(z)_{cls} \quad (2.6)$$

$$A_{w \times h \times 4k}^{reg} = \varphi(x)_{reg} * \varphi(z)_{reg}$$

where w and h are the width and the height of the feature maps and k is the number of anchors. For details regarding RPNs refer to section 2.1.4.

The Siamese Network is pretrained on the ImageNet [10] dataset and the Siamese-RPN is trained end-to-end using Stochastic Gradient Descent (GSD). Therefore template and detection pairs of the same video of the ImageNet video dataset and the Youtube-BB [49] are picked. For the training of the RPN the loss function of “Faster R-CNN” [5] is employed.

Li et al. formulate the tracking problem as a “One-shot learning” problem where the goal is that the template branch in the SN predicts the weights for the two kernels of the detection branch in the RPN.

These two kernels are computed only once (“One-shot”) at the initial frame of the respective video. The features of the current detection image are then convolved with the template kernels to get $A_{w \times h \times 2k}^{cls}$ and $A_{w \times h \times 4k}^{reg}$. After filtering out the top proposals, they list two strategies to select the best fitting bounding box. First, bounding boxes that are located too far away from the center are discarded. This strategy can efficiently remove outliers, because normally the target object does not move far from frame to frame. The second strategy is to suppress displacements by a cosine window and adding a scale change penalty. After this proposal re-ranking Non-maximum-suppression (NMS) is performed to get the final bounding box.

The “SiamRPN” tracker is running on a PC with an Intel i7, 12 GB RAM and a NVIDIA GTX 1060 at 160 FPS which is nearly two times the speed of SiamFC [7]. Li et al. also evaluate their tracker on the VOT-2015 benchmark and they rank at the first place, outperforming SiamFC in the EAO measure by 23% (0.358).

As a further development, the authors of SiamFC tried training their tracker with deeper networks like ResNets [4] but could not observe performance gains. They identified missing spatial invariance of the target objects learned features due to padding as the main reason for that. The learned features have to be spatial invariant because the target object could be located anywhere in the search image. Their proposed architecture SiamRPN++ is based on SiamRPN and able to exploit the benefits of using deep CNNs as feature extractors. SiamRPN++ is described in detail in section 3.1.

Zhu et al. [50] present a novel Distractor-aware Siamese Region Proposal Network (DaSiamRPN) for accurate and robust long-term tracking. Long-term tracking means that the target object can be lost or fully occluded throughout the tracking process. So trackers are required to perform re-detection after the target object reappears. The authors state 3 main problems of previous Siamese tracking approaches:

- Only the foreground is discriminated from the non-semantic background (non-semantic means no real objects, just background).
- Due to simplicity and the fixed-model approach no online model update is possible which is often critical to account for drastic appearance changes.
- They are not able to handle target loss or fully-occluded targets because of their local search strategy.

The main contribution of this paper is a novel distractor-aware offline training strategy. The authors claim that the distribution of training data heavily influences the quality of the tracking network. SiamFC and SiamRPN train their network with images from different frames of the same video. The search area of these images consists mainly of the non-semantic background while semantic entities

and distractors occupy less image area. This is the reason why these networks learn to discriminate foreground from non-semantic background, but lack in learning instance-level representations. The video detection datasets used for the training of SiamFC and SiamRPN are not sufficient for high-quality siamese tracking. So Zhu et al. generate image pairs out of still images from detection datasets (ImageNet detection and COCO) through augmentation techniques (translation, resize, grayscale, et al.). Additionally, they add semantic negative pairs both from the same categories and different categories. This helps the tracker to avoid drifting to arbitrary objects in scenarios like full occlusion or target loss. The authors also present an iterative local-to-global search strategy that can re-detect targets. The distractor-aware module in DaSiamRPN considerably improves the robustness while also operating at high FPS as SiamFC and SiamRPN.

Zhang et al. [51] proposed a method to overcome the problem when predictions are not accurate (below < 0.3), the subsequent predictions may become unreliable as the regression network of the RPN has to deal with weak predictions that were not part of the training dataset. Therefore they present a novel object-aware anchor-free tracker “Ocean” that directly regresses the bounding box coordinates without using anchor boxes. This improves the tracker’s ability to distinguish target objects from complex backgrounds. The results on the VOT2018 benchmark [6] show that the offline Ocean tracker is outperforming SiamRPN++ by 5.3 points in EAO.

2.2.3 Visual Transformers

The results from the latest VOT2022 tracking challenge [44] show that trackers based on SNs or DCFs are outperformed by trackers based on visual transformers. Out of all the trackers, 47% belong to the category of visual transformer-based trackers, 41% use DCFs, and only 6% use classical siamese correlation networks. As for the top trackers, nine of them use transformers as their primary tracking method while one uses deep DCFs. “MS_AOT” is currently the highest-ranked tracker on the public testing set, as measured by EAO. It relies on the transformer-based video object segmentation method called AOT [52].

Transformers [53] were initially developed for creating hierarchical attention-based networks for machine translation. Transformer blocks compute correlation with all the input elements and aggregate their information by using attention mechanisms. Transformer networks have been extensively employed in natural language processing (NLP) tasks. Recently, transformers were applied to various computer vision tasks, such as image classification, object detection and segmentation. They showed promising performance compared to networks based on CNNs.

Vision Transformers were introduced by Dosovitskiy et al. [54]. In contrast to CNNs, input images are considered as a series of patches. Each patch is flattened into a single vector by concatenating the

channels of all the pixels within the patch, and then it is linearly projected to reach the desired input dimension. Raghu et al. [55] explain the differences between the conventionally used CNNs and Vision Transformers (ViTs).

2.3 Optimizing Convolutional Neural Networks For Resource Constrained Hardware

Deep learning algorithms like CNN were conventionally executed in centralized cloud environments with weak resource restrictions [56]. But cloud computing is not always suitable. When it comes to real-time obstacle detection in a car using cameras and a neural network, using cloud computing poses several challenges. How are the data transmitted to the cloud servers? How secure is this transmission? What happens when there is no connection to the cloud? Can obstacles be truly detected in real-time under these circumstances? Executing the networks on embedded devices, where the computation is performed locally, mitigates these issues. Devices can operate offline since all of the computation is done on the device itself. Or the bandwidth of the cloud connection can be reduced by lowering the amount of data that needs to be transmitted. Sensitive data can be processed locally, reducing the risk of data breaches during data transmission. However, a disadvantage is that embedded devices have limited resources.

Based on these requirements, vendors like NVIDIA developed embedded devices for hardware acceleration of deep learning algorithms. Nevertheless, optimizations of these algorithms are required to leverage the potential of the hardware accelerators in order to fulfill real-time requirements.

In this section methods for optimizing a CNN are discussed, whereas details about the chosen embedded device are presented in section 3.2.

2.3.1 Pruning

Pruning CNNs is a widely used technique for reducing the size and computational complexity of a network without sacrificing its performance. It is based on the idea that not all connections in a network contribute equally to its performance. By identifying and removing these redundant connections, the network's size and computational complexity can be reduced without compromising its performance. Pruning can be applied at different levels of a network, including individual neurons, layers, or filters.

Pruning in CNNs can be classified into two main categories. The first is unstructured pruning, which involves removing values independently of their position and based solely on some external metric. A typical metric is the absolute value of the weight. If the weight is smaller than a certain threshold, the parameter is eliminated. The second category is structured pruning, which takes into

account the position and structure of the data within the network. This approach results in a more coarse view of the weight matrix, considering weights only in larger groups such as a filter kernel.

One common approach to pruning CNNs is iterative pruning, which involves training a network, pruning the connections that have the least impact on the network's performance, and then retraining the network. This process is repeated until the desired level of compression is achieved. It is essential to balance the compression ratio with the network's performance. However, pruning can even improve the network's performance by reducing overfitting and improving generalization.

Unstructural Pruning

In their work, Han et al. [57], proposed a straightforward approach for pruning by using a strong L2 regularization term during fine-tuning and eliminating parameters with values below a pre-defined threshold. This is highly effective in reducing network size and demonstrates good performance. However, it is worth noting that network compression may not necessarily result in faster inference, as modern hardware often leverages computation regularities for high throughput.

In the case of convolution, this technique involves setting individual values within a filter kernel to zero. As a result, higher pruning rates can be achieved compared to structured pruning. However, the advantages of this approach are limited. This is because each value must be constantly checked for zero, and zero entries must be skipped, which can be computationally expensive. The simple removal of filter weights is not possible due to destroying the general matrix multiply form, which is used in libraries like PyTorch.

Structural Pruning

Compared to unstructured pruning, structured pruning allows for a more coarse form of pruning, removing larger sections of the network that can be excluded from computation. In contrast to unstructured pruning, structured pruning leaves the matrix intact and does not potentially break the general matrix multiply form. This is beneficial for certain hardware architectures, making it a more hardware-friendly approach. Structured pruning selectively removes weights while preserving the network's overall structure. It removes larger sections of the network by targeting groups of weights rather than individual weights. These groups can include filter kernels, entire channels, or entire layers. The position and structure of the weights are taken into account when making pruning decisions.

Filter-level pruning involves discarding the entire filter if it is deemed unimportant. The effectiveness of this pruning method heavily relies on the accuracy of the applied importance evaluation criterion. Criteria like calculating importance scores with L1 norm [58] or calculating filter importance based on statistics computed from its next layers [59] are used.

Numerous filter-level pruning methods have been proposed, but pruning residual connections in ResNets remains difficult. One popular method is to prune only filters inside the residual connection, leaving the number of output channels unchanged. This leads to an hourglass structure replacing the original bottleneck. However, this structure severely limits the representation ability of the middle layers. Therefore, to speed up networks, it is better to prune channels both inside and outside the residual connection. This results in a pruned block that still retains the bottleneck shape or opens up like a wallet.

Luo et al. [9] propose a method named CURL that prunes both channels inside and outside the residual connections via a KL-divergence-based criterion. Inspired by ThiNet [59], CURL can significantly outperform previous state-of-the-art pruning methods on ImageNet. Assuming that there are two residual blocks in the present stage (Figure 2.14), each comprising three convolutional layers (including batch normalization and ReLU activation), a down-sampling layer is typically required to handle varying activation sizes and channel numbers between stages. Since shortcut connections are present, the channel numbers of all residual blocks within a stage must remain consistent for the summation operations to be valid. One intuitive approach is to eliminate the output channels of the residual block

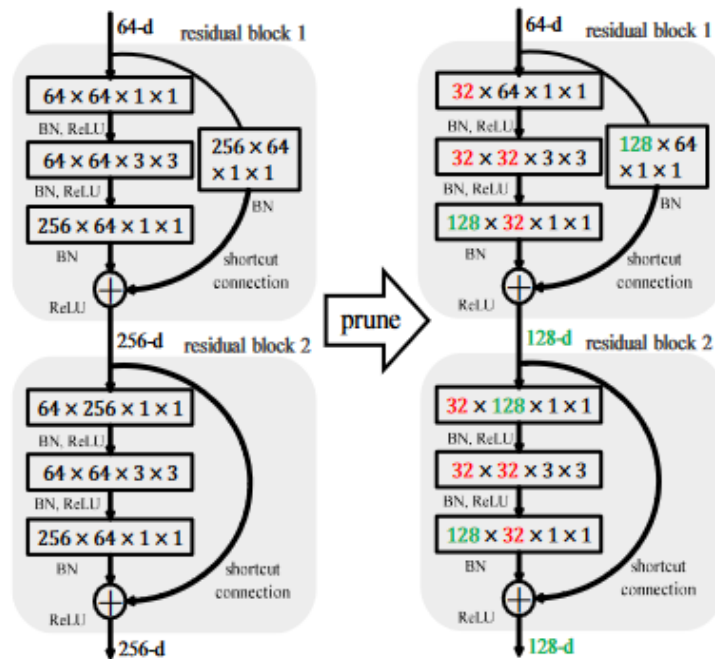


Figure 2.14: CURL [9]: Not only channels inside the residual block (red) are pruned, but also output channels (green). In each rectangle (representing a convolutional block), the first two numbers indicate the output channels and input channels, respectively.

one by one and assess the corresponding loss of information. However, since the structure is constrained, the output channels of each residual block must be removed simultaneously. CURL therefore

leverages the batch normalization layer [24], which has been adopted by most modern CNNs. A batch normalization layer performs the following transformation on a mini-batch B :

$$y_i = \gamma \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta \quad (2.7)$$

where γ and β are the trainable parameters, and μ_B and σ_B denote the mean and the standard deviation of input activations over B . $\gamma = \beta = 0$ yields in zeroing out the corresponding output channel since the batch normalization is channel independent. Before and after channel removal the performance is evaluated. The KL-divergence is a popular method to compare the output probability p of the original network with the output probability q of the network with channel removal:

$$s = D_{KL}(p||q) = \sum_{n=1}^n p_i \log\left(\frac{p_i}{q_i}\right) \quad (2.8)$$

When the current filter is redundant, the value of s will tend towards 0. In such cases, removing this filter will have no significant effect on the prediction results. The process will be iterated 256 times (for a residual block depicted in Figure 2.14), producing 256 importance scores, each corresponding to a channel.

Regarding the channels located in the residual block (indicated by red numbers in Figure 2.14), one filter of the current layer is removed at each step. Subsequently, the filter scores of all layers are sorted in ascending order, and the top k filters are eliminated, resulting in a pruned, compact model. To avoid any adverse effects, such as having very few filters left in a layer after pruning, the minimum compression rate of each layer should not fall below a certain threshold.

A pruned and fine-tuned ResNet50 using CURL achieves a Top-1 accuracy of 73.39% on ImageNet with 6.67M parameters - state-of-the-art Top-1 accuracy of an original ResNet with 25.56M parameters is 76.15%. CURL significantly outperforms previous state-of-the-art pruning methods [9].

2.3.2 Quantizing

The idea of quantization is to represent the weights and activations of CNNs with lower bit widths. Instead of using 32-bit floating-point numbers, quantization may use floating-point numbers. While this may seem like a significant reduction in precision, in practice, it can achieve a high level of accuracy while reducing the model size, memory requirements, and computation cost.

Quantizing CNNs can be achieved through various methods, including uniform quantization, which maps values linearly to a specific number of levels. And non-uniform quantization, which maps values to varying levels based on their probability distribution. Moreover, hybrid approaches that incorpo-

rate both uniform and non-uniform quantization techniques are also available. Despite its advantages, quantization is not without its challenges. These include precision loss in quantized values and the challenge of selecting the optimal bit width for each layer. Additionally, the non-linear nature of activation functions in CNNs can infer difficulties in maintaining accuracy during the quantization process.

The HashNet approach, as proposed in [60], involves quantizing the network weights by grouping them and sharing their values within each group before training. This results in only the shared weights and hash indices needing to be stored, leading to significant savings in storage space. Another technique proposed in [57] uses an enhanced quantization method in a deep compression pipeline, which achieves compression rates of 35x to 49x on AlexNet [15] and VGGNet [61]. However, while these techniques can significantly reduce storage space, they do not necessarily save runtime memory or inference time. This is because during inference, shared weights must be restored to their original positions. Rastegari et al. [62] and Courbariaux et al. [63] even quantize weights into binary or ternary weights, which allows them to decrease the model size greatly and significantly speed up the inference time. However, the use of such an aggressive low-bit approximation method often results in a moderate loss of accuracy.

2.3.3 Knowledge Distillation

Knowledge distillation transfers knowledge from a large, complex model into a smaller, more compact one by using a teacher network's output to train the smaller student network. The student network learns to minimize the difference between its output and the teacher network's output. One of the main advantages is to transfer knowledge from a pre-trained network (e.g. on ImageNet) into a smaller network, which reduces the amount of training data required and improves the speed of training.

Caruana et al. [64] introduced the concept of selecting a diverse set of base models from a pre-existing library and combining them into an ensemble. The selection process is done by optimizing an objective function that balances ensemble accuracy with ensemble diversity. Based on this paper, Hinton et al. [65] introduced the teacher-student method.

Chapter 3

Methodology

3.1 SiamRPN++ [1]

As already mentioned in section 2.2.2, Li et al. were the first to exploit deep neural networks in siamese networks by introducing SiamRPN++. The primary finding of this work is that employing more deep and sophisticated networks (like ResNet [4]) can greatly enhance the effectiveness of the siamese network-based tracking algorithm. However, training the siamese tracker using deeper networks with previous methods (like in SiamRPN [8]) does not produce the anticipated enhancement in performance. In the following, SiamRPN++ is described in detail.

A visual object tracker is required to be translation invariant, which is the ability to ignore positional shifts or translations of the target object in the image. In other words, the tracker shall be able to locate the object regardless of where it appears in the current frame. Let z be the current target object and x the current search frame, then the siamese network-based tracking algorithm tries to learn the tracking similarity f :

$$f(z, x) = \phi(z) * \phi(x) + b \tag{3.1}$$

ϕ denotes the feature extractor and b models the offset of the similarity value. Strict translation invariance can then be formulated as $f(z, x[\Delta\tau_j]) = f(z, x)[\Delta\tau_j]$ where $\Delta\tau_j$ is the translation shift sub window operator.

Li et al. argue that padding, which is inevitable in deep networks, destroys the strict translation invariance. In networks with padding, a strong center bias is learned during training. To overcome this problem, they introduce their spatial aware sampling strategy for training. Target objects in the center of training images are shifted by values generated by a uniform distribution. Figure 3.1 visualizes the aggregated probabilities of target positions generated on a test dataset. Applying zero shift leads to a strong center bias with border values degraded to zero. Increasing the shift value prevents the

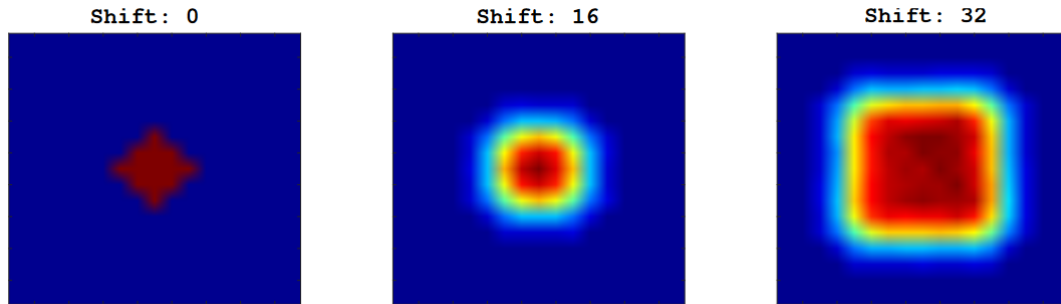


Figure 3.1: Target position probabilities when using random shift. [1]

network from learning this strong center bias. The target position probabilities are closer to the location distribution of test objects.

In a more recent work, Chaman et al. [66] claim that downsampling or stride is the reason for translation invariance. Data augmentation techniques like the spatial aware sampling strategy in SiamRPN++ training improve the robustness of the network over shifts in the image. However, this is only valid for images seen during training the network. Therefore they propose a non-linear downsampling strategy to make CNNs fully shift-invariant.

Li et al. developed the SiamRPN++ tracker with focus on using ResNet-50 as backbone network. As the original ResNet-50 is not ideal for dense siamese network prediction, they modify the conv4 and conv5 blocks to have a unit spatial stride (Figure 3.2). This reduces the effective strides from 16 pixels and 32 pixels to 8 pixels. To reduce the number of channels to 256 for each siamese RPN block, extra 1×1 convolution layers are added. Although the paddings of all layers were maintained, the spatial size of the template feature increased to 15, resulting in a heavy computational burden on the correlation module. To address this, the template feature is cropped around the center to a 7×7 region. In contrast to traditional siamese methods, the parameters of the ResNet-50 are trained jointly in an end-to-end manner. The layers of a ResNet network have different significance levels as their receptive fields vary significantly. Features from early layers focus on low-level information, such as color and shape. These are essential for localization, but they lack semantic information. Features from later layers have richer semantic information, which can be advantageous in challenging scenarios such as motion blur or deformation. The outputs of the three siamese RPN modules are combined via weighted sum as their output sizes have the same spatial dimensions.

The cross-correlation in SiamRPN [8] incorporates much higher level information through the addition of a large convolutional layer (UP-Xcorr). However, the significant increase in parameters in the up-channel module has resulted in an imbalance in the parameter distribution. Specifically, the RPN module now contains 20M parameters, while the feature extractor only has 4M parameters. This parameter imbalance has made it difficult to optimize training in SiamRPN. To overcome this issue, Li

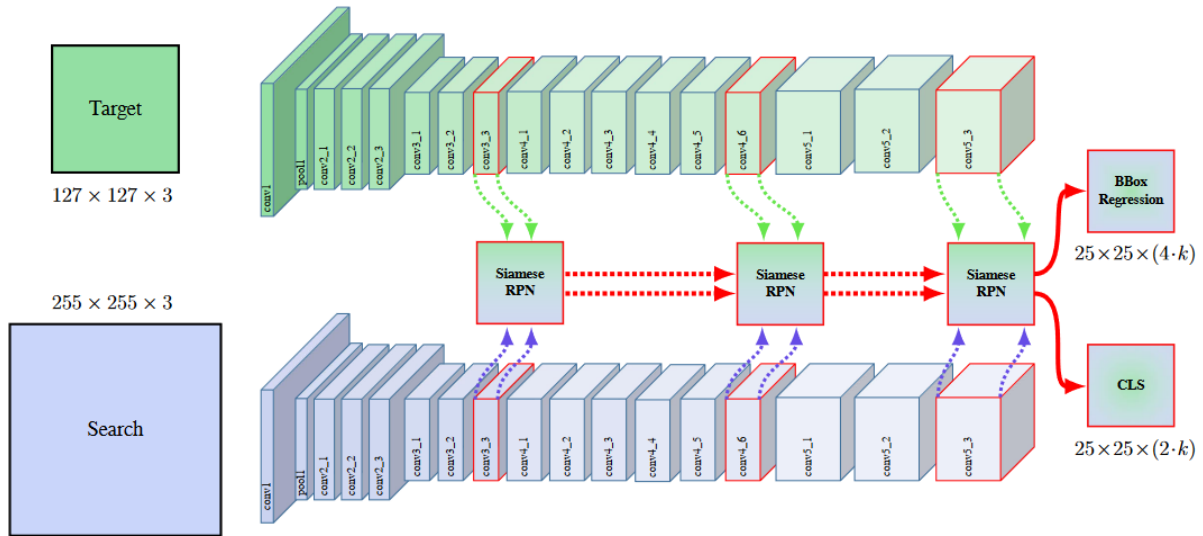


Figure 3.2: Network architecture of SiamRPN++ [1]. The outputs of three Siamese RPNs, which take features from different depth levels, are fused into a output prediction.

et al. present a more lightweight cross-correlation layer (Depthwise Cross Correlation) with 10 times fewer parameters. Switching from cross-correlation to depthwise correlation can significantly decrease computational costs and memory usage. This adjustment balances the number of parameters between the template and search branches, leading to a more stable training process.

SiamRPN++ surpasses all other trackers submitted at the VOT 2018 challenge [6]. It outperforms the second-ranked tracker by 2.5%. The authors achieved real-time speed (35 FPS) on an NVIDIA Titan Xp GPU. SiamRPN++, despite being the best-ranked tracker, still has a disadvantage shared by many siamese trackers, which is weaker robustness compared to state-of-the-art correlation-based filter methods that use online updating to adapt to templates.

Li et al. state that the selection of the feature extractor (backbone network) plays a critical role since the number of parameters and the types of layers directly impact the tracker's memory usage, processing speed, and overall performance. Consequently, the backbone architecture is suitable for optimizations.

3.1.1 Training

The ResNet-50 backbone is initially trained on ImageNet [10]. The overall network (see Figure 3.2) is trained on the training datasets of COCO [67], ImageNet DET [10], ImageNet VID and YouTube-BoundingBoxes [49]. Target images are 127×127 pixels and search images are 255×255 during training and testing.

Before training, some preparations need to be made. Initially, the raw images and annotations of the

datasets must be downloaded and unpacked. This process can be quite time-consuming, particularly in the case of YouTube BoundingBoxes, which comprises around 380,000 videos with an average duration of 19 seconds. Then the training images for the target branch and the search branch (Figure 3.2) have to be generated:

- Load the ground truth bounding box and calculate the center position and size of the bounding box.
- Increase the size of this bounding box by a contextual factor (e.g. 0.5)
- Create a square search window around the target position (for both branches).
- Perform scaling (linear transformation) via affine transformation and crop out the search window.

Figure 3.3 illustrates examples for generating target and search images out of a single training image.

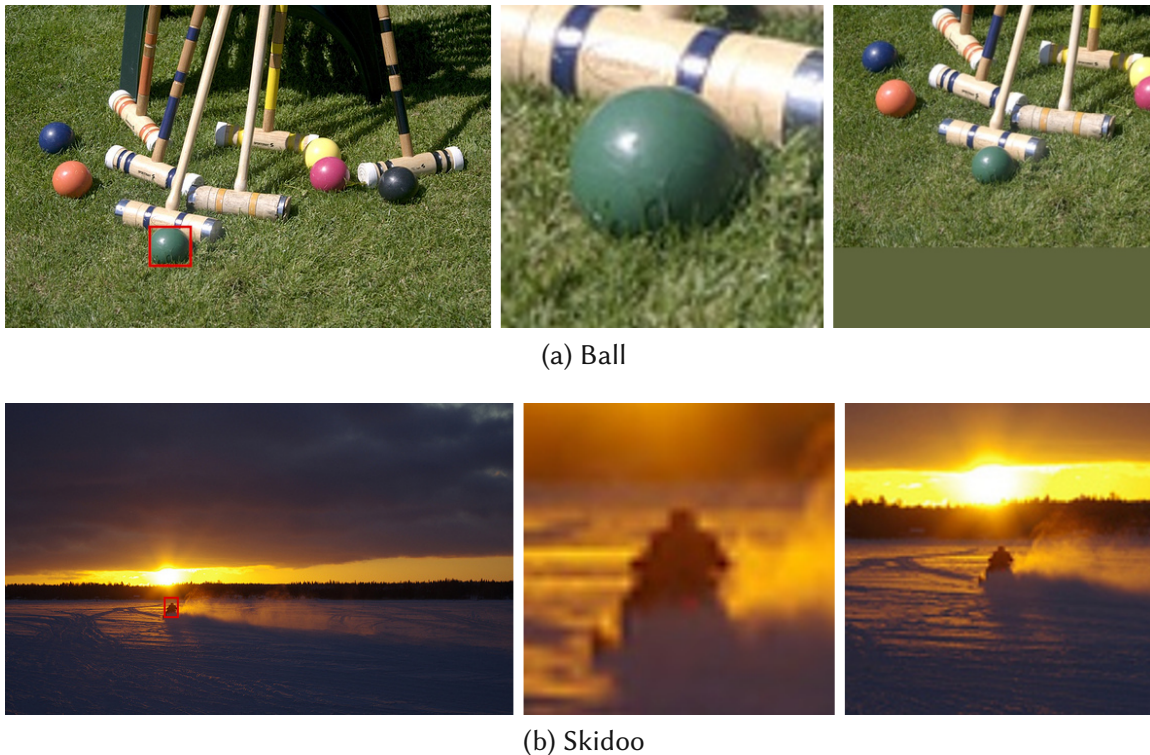


Figure 3.3: Training images (left) from ImageNet DET [10] with their target (middle) and search (right) crops.

The next step is to configure the training hyperparameters like batch size, initial learning rate and number of epochs. Then the PyTorch model of the SiamRPN++ network architecture is instantiated and filled with the ResNet-50 backbone weights (either from the original backbone or a pruned backbone).

3.2 Hardware Setup

The objective of this thesis is to optimize the SiamRPN++ [1] tracker for a resource-constrained hardware platform. The NVIDIA Jetson platform offers a great range of devices for embedding computing and NVIDIA claims that Jetson is the world's leading platform for autonomous machines and other embedded applications. Jetson devices are equipped with a CUDA-capable GPU which is used to accelerate complex machine learning tasks [56].

NVIDIA Jetson devices are designed for a wide range of applications, from autonomous robots and drones to smart cameras and industrial IoT. They are built to deliver high-performance processing capabilities for running deep learning models and computer vision algorithms in real-time. The Jetson device family includes four main models, including the entry-level Jetson Nano, the mid-range Jetson TX2 and Jetson AGX Xavier, and the high-end Jetson Orin. All devices feature NVIDIA's powerful GPU architecture, which enables parallel processing and efficient neural network inference. They also come with a variety of ports and connectors, including USB, Ethernet and HDMI for connecting cameras and sensors. Software libraries and development tools are provided to enable developers the creation and deployment of AI applications.

For all current members of the Jetson platform refer to documentation from NVIDIA ¹. This thesis focuses on optimizing the tracker on the NVIDIA Jetson AGX Xavier (Figure 3.4) with 16GB RAM. All further measurements and results relate to executing the tracker on this specific hardware platform.



(a) NVIDIA Jetson Xavier AGX standalone module (b) NVIDIA Jetson Xavier AGX Developer Kit

Figure 3.4: NVIDIA Jetson Xavier AGX

The Jetson Xavier AGX 16GB is equipped with an eight-core NVIDIA Carmel Arm CPU, a 512-core NVIDIA Volta GPU, and 16GB of LPDDR4x memory, providing high performance and memory bandwidth. Users can configure the operating modes between 10 W, 15 W, and 30 W, enabling the

¹<https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/>

Jetson Xavier AGX to achieve more than 20 times the performance and greater than 10 times the energy efficiency of its predecessor, the Jetson TX2. The Jetson AGX Xavier Developer Kit (Figure 3.4b) comes fully equipped with all the necessary components and JetPack software. It comes pre-assembled with the Jetson AGX Xavier compute module (Figure 3.4a), an open-source reference carrier board, a cooling solution, and a power supply. Additionally, the developer kit includes several convenient I/O ports.

Training CNNs requires tremendous computational power as it is the most resource-intensive task. Modern GPUs are well suited to handle this kind of task. Supercomputers, equipped with clusters of GPUs, provide even more resources to efficiently these networks. The Vienna Scientific Cluster (VSC) provides a supercomputer located in Vienna, Austria. The VSC provides a state-of-the-art computing infrastructure to researchers, enabling them to perform simulations and data analysis for their scientific work. It is equipped with a range of computing resources, including clusters of high-performance computing nodes, each containing multiple CPU and GPU cores. The VSC-5 with its GPU nodes is used for training and pruning (Reference to section?) the networks used in this thesis. The GPU nodes are equipped with two NVIDIA A100 cards (40GB memory each).

3.3 Evaluation Metrics

Docker containers allow the creation of a clean and consistent environment that is free from conflicts with other software dependencies or system configurations. All measurements presented in section 4 are performed in a docker container as this provides a high level of reproducibility and consistency that is often difficult to achieve with traditional software development and deployment methods.

NVIDIA provides several development and deployment containers for Jetson devices. In this work, the “L4T-Base container image“ of the 32.6.1 release, which mounts platform-specific libraries into the l4t-container from the underlying host, is used. Most important, the TensorRT library (version 8.2.1.8) from the Xavier (running with JetPack 4.6.2 and CUDA 10.2) is mounted into the container. Additionally, the PyTorch framework of version 1.9.0 is installed inside the container.

3.3.1 Evaluation protocols and performance measures

The VOT challenge divides tracking algorithms into two classes: short-term and long-term trackers. Short-term tracking assumes that trackers cannot successfully re-detect the target once it is lost, and therefore they are reset after such an event. Resetting the tracker in case of failures removes target predictions that are irrelevant for tracking accuracy. In long-term tracking, trackers are expected to re-detect the target after it has been lost and are not reset after such an event.

In the VOT reset-based evaluation protocol (for short-term tracking), accuracy and robustness are

the main performance measures. Accuracy is defined as the average Intersection over Union (IoU) (Equation 3.2 and Figure 3.5) over the testing dataset. It describes how well the bounding box predicted by the tracker overlaps with the annotated ground truth bounding box. The robustness measures how many times the tracker lost the target (fails) during tracking. A failure is indicated when the IoU between predicted and ground truth bounding box is below a certain threshold (e.g. 0).

$$\text{IoU} = \frac{\text{area of intersection of two bounding boxes}}{\text{area of union of two bounding boxes}} \quad (3.2)$$

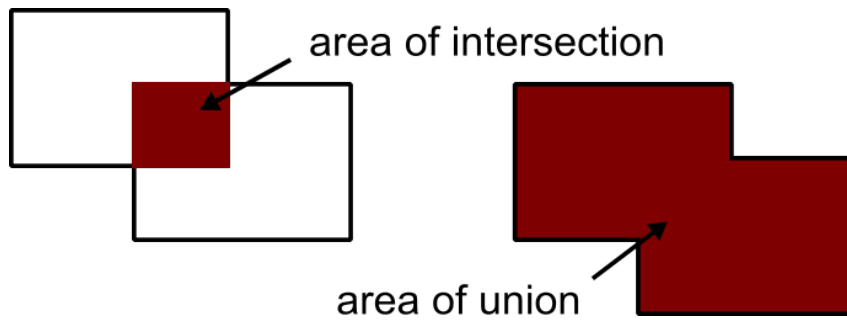


Figure 3.5: IoU: Illustration of notations from Equation 3.2

Beginning in 2015, the VOT challenge has adopted the Expected Average Overlap (EAO) as its primary measure, which is a combination of accuracy and robustness in a principled manner. Previously, the average between accuracy and robustness was used to rank proposed trackers. The authors of the VOT challenge claim that the common average of accuracy and robustness can not be interpreted in terms of a concrete tracking application result [12]. Because the VOT shot-term protocol resets the tracker after each failure, potentially multiple segments of a test video are generated. These segments from all test videos can be utilized to compute the EAO. Segments that are shorter than N_S frames and did not finish with a failure are removed. The remaining segments are converted to tracking outputs that are N_S frames in length by either trimming or padding with zero overlaps.

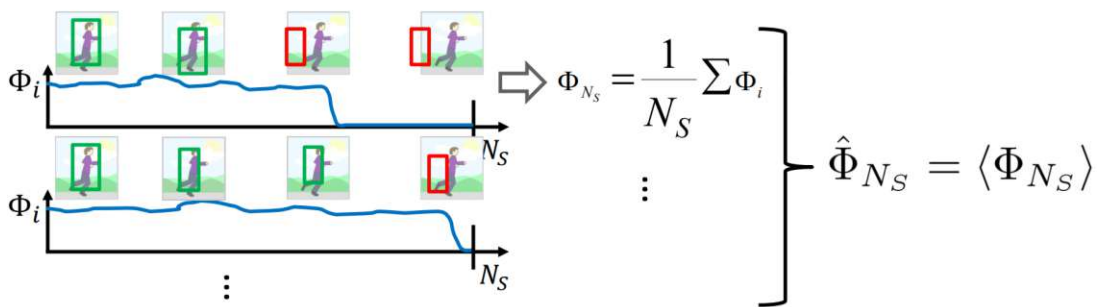


Figure 3.6: Expected average overlap on a N_S frames long sequence [11].

At the start of each segment, the tracker is initialized and left to track until the end. For each

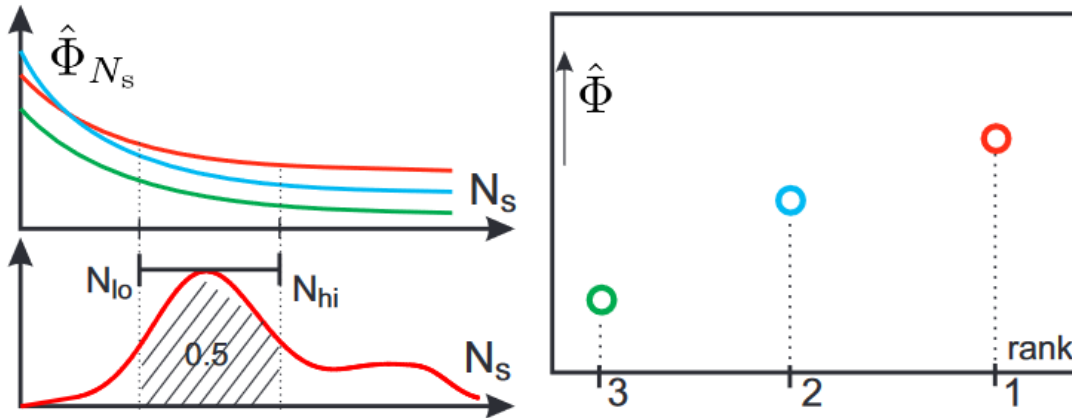


Figure 3.7: Left up: the Expected Average Overlap (EAO) curve. Left bottom: the sequence length pdf. Right: plot of the EAO [12].

segment, an average overlap Φ_{N_S} is calculated (Figure 3.6), and the average of all segment averages gives the estimate of $\hat{\Phi}_{N_S}$ (Equation 3.3 and Figure 3.7). By computing this for various values of N_S , we can generate an estimation of the expected average overlap curve (Figure 3.7). The EAO measure is calculated by averaging the EAO curve over an interval $[N_{lo}, N_{hi}]$ of typical short-term segment lengths (Equation 3.3). The typical short-term segment lengths $[N_{lo}, N_{hi}]$ are estimated as the mode of the probability density function over the segment lengths.

$$\hat{\Phi} = \frac{1}{N_{hi} - N_{lo}} \sum_{N_S=N_{lo}:N_{hi}} \hat{\Phi}_{N_S} \quad (3.3)$$

One of the main distinctions between short-term and long-term trackers is that long-term trackers must be capable of handling situations where the target is absent from view for extended periods. Therefore, a common evaluation protocol for long-term tracking is a no-reset protocol. Performance measures in long-term tracking are varied and have not been standardized as in short-term tracking. Previously performance measures from object detection literature were used, with the primary metrics: precision, recall, and F-measure calculated at a 0.5 IoU (overlap) threshold. But 0.5 turned out to be too restrictive for tracking. To overcome this, Lukezic et al. [68] introduced tracking precision, tracking recall and tracking F-measure which are all independent from an IoU threshold. Their main measure, the tracking F-measure, has been demonstrated to convert into a typical short-term measure (known as "average overlap") when computed in a short-term configuration. The VOT uses these three measures for evaluating long-term trackers.

3.3.2 Testing Dataset

Since the beginning of the VOT challenge in 2013, also testing datasets for the various sub-challenges have been made available. Initially, they used videos, which have already been used in the tracking community. Each video was additionally labeled with five visual attributes: occlusion, illumination change, motion change, size change, and camera motion.

In addition to the VOT challenge testing datasets, there are other datasets. SiamRPN++ evaluates its tracker on the OTB-2015 [69], UAV123 [70], LaSOT [71], and TrackingNet [72] datasets. OTB-2015 refers to the standardized Object Tracking Benchmark (OTB) and focuses on robustness. The UAV123 dataset provides 123 sequences as a benchmark for low-altitude UAV (unmanned aerial vehicle) tracking. LaSOT offers high-quality, dense annotations on a large scale, comprising 1,400 videos in total, with 280 videos included in the testing set. TrackingNet is a large-scale dataset and benchmark for object tracking in the wild.

This thesis focuses on the VOT2018 short-term tracking challenge testing dataset consisting of 60 videos and 21356 single images as the evaluation benchmark. SiamRPN++ uses the VOT2018 testing dataset and in order to be able to compare results, the VOT2018 testing dataset was also used here.

3.3.3 Inference latency

Inference latency is the time delay between sending data to a machine learning model for processing and receiving the output. It is a critical performance metric for real-time applications that require quick and accurate predictions. Inference latency is affected by several factors, such as the complexity of the model, the hardware infrastructure used to run the model, and the size of the input data. An approach for measuring inference latency is to use profiling tools that provide detailed information about the performance of the model, such as the amount of time spent in each layer of the model or the amount of memory used during inference.

In this thesis, the evaluation of the tracker on the test dataset includes the measurement of inference latency. The measurement scheme for inference latency adopted from PySOT involves measuring the time it takes for the tracker to report the bounding box after the current image of the current test video is loaded. To evaluate the inference latency in the context of a video, the inference latencies for all images in the video are divided by the total number of images to obtain the Frames Per Second (FPS) score. The FPS score represents the average number of frames that the model is able to process per second, and it is a useful performance metric for applications that require real-time processing of video data.

When evaluating short-term tracking performance, it is important to consider the reset mechanism

that is used when the target object is lost. The reset mechanism is designed to re-detect the target object and reinitialize the tracker when it is lost, but this can cause distortions in the FPS scores. When the tracker loses the target object, the FPS score appears higher because the tracker is skipping 5 frames. This should be carefully considered when interpreting the results of such evaluations.

3.3.4 Power consumption

Power measurement is an important aspect of evaluating the performance of machine learning models on hardware platforms such as the NVIDIA Xavier. Aside from using a power meter, power consumption can also be measured using software tools such as Tegrastats. Tegrastats is a system monitoring tool that is included with the NVIDIA JetPack SDK and provides real-time monitoring of various system parameters including CPU, GPU, and memory usage, as well as power consumption. Additionally, Tegrastats can provide insights into the temperature and utilization of various system components, allowing for a more comprehensive understanding of the performance of the device. The reported power consumption from Tegrastats refers to the total power consumed by the system - not just the tracking process itself. However, this is irrelevant for comparing different optimizations, as the power consumption values are always referenced to the baseline.

Tegrastats is started directly before the testing dataset is loaded and stopped as soon as the last video in the dataset has been processed. Once the testing is complete, the generated output from Tegrastats can be parsed to obtain the power consumption values. Afterwards the power consumption values can be visualized.

The NVIDIA Xavier supports several power modes, each with its own trade-offs between performance and power consumption. These modes conclude: MAXN, 15W, 10W and 5W. As the main focus of this thesis focuses is on optimizing the tracking speed, only the MAXN power mode is used. MAXN is the highest performance mode and provides the maximum computing power of the device.

3.4 Conversion into ONNX

The SenseTime Video Intelligence Research team provides a Python implementation of SiamRPN++ in their PySOT repository ². They utilize the popular Pytorch framework, which is an open-source machine learning framework developed by Facebook's AI Research (FAIR) team. With PyTorch, developers can use an API to design and train neural networks, which can be accelerated using either CPU or GPU. PyTorch also includes a variety of built-in functions for various operations such as convolution, pooling, activation, and loss calculation. PyTorch's dynamic computational graph is a major benefit,

²<https://github.com/STVIR/pysot>

as it enables developers to make real-time modifications to their models during training. PySOT also features a toolkit for evaluating trackers on common tracking datasets (e.g. VOT, LaSOT, OTB).

The Open Neural Network Exchange (ONNX) is an open format for representing machine learning models. It supports cross-functionality between various machine learning frameworks (e.g. PyTorch, Tensorflow, Keras). As already mentioned, e.g. PyTorch can be used to train a CNN and the resulting model is saved in a specific data format (.pth). Especially in the domain of resource-constrained embedded hardware it can happen that PyTorch is not supported, but the trained model has to be deployed on the target platform. In this case, Open Neural Network Exchange (ONNX) acts as an intermediate model format because it allows to train a model with one framework and deploy it using another framework. The frameworks typically provide a conversion method to export the model in the ONNX format.

Section 3.1 already mentioned that the spatial size of the target (template) features increased to 15 as a consequence of maintaining padding in all layers. In order to avoid resulting heavy computational burdens, the target features are cropped around the center to a 7×7 region. The following describes how this circumstance affects the conversion into the ONNX format.

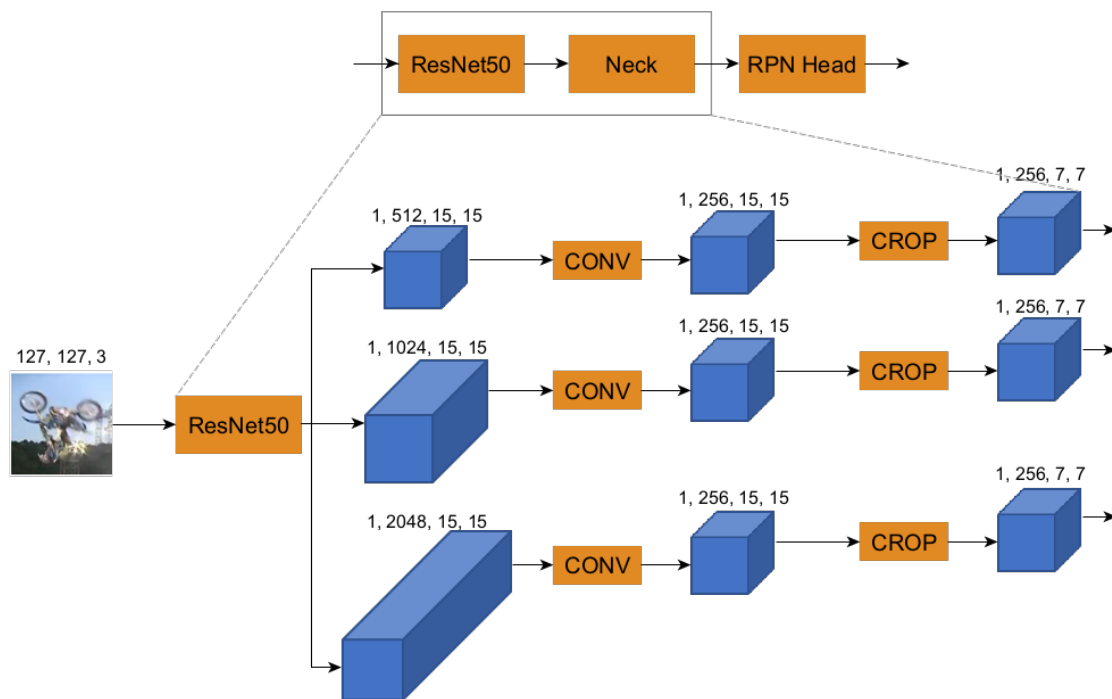


Figure 3.8: Illustration of downsampling and cropping template features in the neck section of the network.

Figure 3.2 shows that the outputs of three RPNs that take ResNet-50 backbone features from three different depth levels are fused into a single output prediction. In order to fuse these outputs into one single output, the channels of the three different features have to be equal. Therefore, the so-called

"neck" section of the network downsamples the outputs via 1×1 convolutions to 256 channels. The target features serve as kernels for the depthwise Cross-correlation in the RPN head module. Because 15×15 kernels require high computational effort, the target features are cropped around the center region to obtain 7×7 feature maps.

However, cropping only affects the target features, not the search features. The PyTorch implementation checks the third dimension of the target features via if statement after the downsample convolution. If it is less than or equal to 20, cropping around the center region is applied. The ONNX framework also provides an if operator, but I decide to create separate networks for computing target and search features. The reason for this is that TensorRT can not handle dynamic networks (see section 3.5).

Figure 3.9 illustrates the introduced network structure. After computing target features and search features with the backbone (ResNet-50), the adjustment blocks (downsample convolutions) are applied on these features. I have combined these two steps into one network each: "target network" and "search network". Now a third network is necessary to perform the depthwise cross-correlation and the convolutions in the head modules. This network is named "xcorr network". The results from the "target net" serve as kernels for the depthwise cross-correlations. The output of the "Box Head" module is used to compute the resulting bounding box via bounding box regressor B. And the output of the "Cls Head" is used to compute the classification score (confidence) of the predicted bounding box.

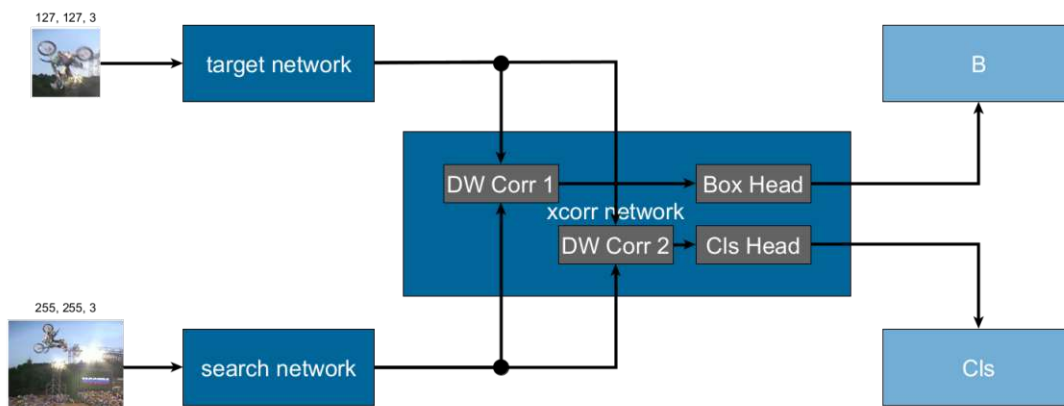


Figure 3.9: Introduced structure to separate SiamRPN++ [1] into three networks.

In the following the workflow to convert a pre-trained PyTorch-SiamRPN++ network into an ONNX network is described. First, the structure of each network is created according to the plan mentioned above. Then, the weights from the pre-trained network are loaded into the three networks. PyTorch provides a function for exporting a network in the ONNX format. It executes the network and records a trace of the operators used. The input tensor must be provided to the function in order to execute

the network. The values in the tensor can be random as long as they are the correct type and size. It is important to note that the input size will be constant in the exported ONNX graph for all input dimensions unless explicitly defined as dynamic. The batch size is fixed at 1, as no other batch sizes are needed for tracking with SiamRPN++.

3.5 Optimization with TensorRT

Developed by NVIDIA, TensorRT is a high-performance library for deep learning inference. It is specifically designed to optimize and accelerate neural network inference on NVIDIA GPUs. TensorRT optimizes the computation graph of a pre-trained neural network to improve its performance, reduce memory usage, and decrease latency. It provides the following optimizations³:

- **Weight quantizing:** To determine the optimal precision for each layer of a neural network, TensorRT runs a calibration process. By decreasing the precision down to 16-bit floating point (FP16) or 8-bit integer (INT8) of specific layers, such as activation and weight layers, TensorRT can improve performance significantly without compromising accuracy.
- **Kernel Fusion:** TensorRT fuses multiple operations into a single kernel, reducing memory access and increasing computation efficiency.
- **Dynamic tensor memory:** TensorRT uses dynamic memory allocation to reduce memory usage and increase efficiency. This allows TensorRT to allocate memory only when it's needed, rather than pre-allocating all memory at the start of the computation.
- **Layer and tensor optimizations:** TensorRT performs several optimizations at the layer and tensor levels, including kernel selection and layout optimization.
- **Mixed-precision computations:** TensorRT provides mixed-precision computations, resulting in faster computation times, reduced memory usage while maintaining accuracy.
- **Multi-Stream Execution:** TensorRT is capable of processing multiple input streams in parallel.

TensorRT operates in two phases: the first phase, which is typically performed offline, involves providing TensorRT with a model definition that it optimizes for a target GPU. In the second phase, the inference with the optimized model is performed. The result of the first phase is the so-called "engine". An engine is created by the TensorRT-builder in a serialized form known as a plan, which can be deserialized or saved for future use. Engines generated by TensorRT are unique to the version of TensorRT utilized to create them, as well as the GPU on which they were constructed. So building an engine must take place on the actual target platform, which is Xavier for this thesis.

³<https://developer.nvidia.com/tensorrt>

Since TensorRT adapts and executes the optimizations individually for each pre-trained network, the generated engine must be regarded as fixed - without the possibility of subsequent changes to e.g. the weights. But during engine building, it is possible to indicate weights, that will be modified later during runtime. This feature called refitting is necessary to update the depthwise cross-correlation of the "xcorr" engine with the results from the "target" engine. But it is important to note that only the previously declared weights can be modified - the structure has to remain.

TensorRT provides C++ and Python APIs with nearly identical capabilities. In this thesis, the Python API is used (for details refer to chapter 4).

Post Training Quantization using Calibration

Post Training Quantization (PTQ) is performed after a high-precision (e.g. FP32) model has been trained. TensorRT provides Post Training Quantization (PTQ) for neural networks and represents quantized floating point values with 8-bit integers. The chosen quantization scheme for both activations and weights is symmetric uniform quantization. The specific quantization levels used for the activations are determined by the selected calibration algorithm, which aims to identify the optimal scaling factor s that balances the trade-off between rounding error and precision error for the given data.

Calculating s for symmetric uniform quantization is described with equation 3.4:

$$s = \frac{\max(|x_{min}|, |x_{max}|)}{127} \quad (3.4)$$

and the whole quantization operation is described with equation 3.5:

$$x_q = \text{roundWithTiesToEven}(\text{clip}(\frac{x}{s}, -128, 127)) \quad (3.5)$$

x_q is the quantized value in the range $[-128, 127]$ and x is the actual value of the weight or activation. "roundWithTiesToEven" is rounding floating point numbers exact in the middle between two integer numbers ("ties") towards the even number.

For PTQ, TensorRT computes a scale value for each tensor in the network. This process is called calibration and requires representative input data. Tensors are the fundamental data structure in TensorRT representing the output or intermediate activations of a neural network layer. Tensors are used to propagate data through the neural network. TensorRT uses the representative input data ("calibration dataset") to collect statistics for each activation tensor. The amount of input data required for calibration is application-specific, but according to NVIDIA, 500 images are sufficient for calibrating ImageNet classification networks. Choosing the best scale value is not a straightforward process be-

cause it involves balancing two sources of error: discretization error and truncation error.

The TensorRT PTQ workflow is illustrated in figure 3.10. The first step is to provide the neural network in ONNX format and the calibration dataset. The calibration dataset should consist of representative and real input data. The process of quantizing the weights of a neural network is comparatively easier than quantizing the activations. This is because weight tensors are directly accessible, and their distributions can be easily measured. In contrast, quantizing the activations is a more challenging task. This is because activation distributions are dependent on the input data, and they cannot be determined using weight tensors alone. Instead, real input data must be used to measure the distributions accurately. Therefore, the high-precision model is evaluated on the calibration dataset to collect statistics about the distribution of activations. Afterwards, the scales from equation 3.4 can be calculated and the actual quantizing operation (equation 3.5) is performed. The result is a quantized TensorRT engine.

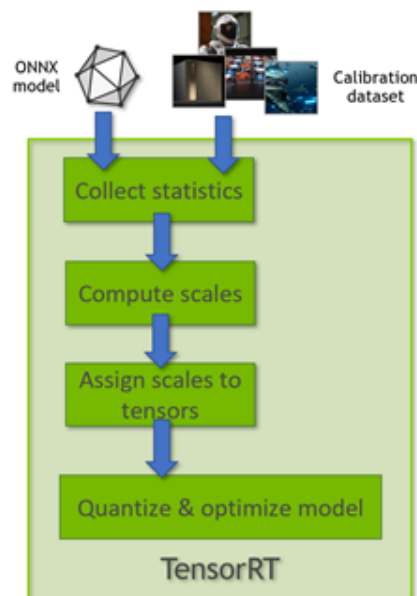


Figure 3.10: TensorRT PTQ workflow [13]

3.6 Pruning the Backbone

In section 3.1 it is mentioned that the ResNet-50 backbone of SiamRPN++ is well suited for optimizations. Pruning is the chosen optimization technique because it offers great speed improvement while maintaining performance. The following describes how the ResNet-50 of SiamRPN++ is pruned and what the workflow looks like.

3.6.1 ResNet-50 of SiamRPN++

Since ResNet-50 is a common CNN architecture, researchers already addressed pruning these types of networks. A wide range of already pruned ResNet-50 networks are published and available for deployment. But the architecture of a ResNet-50 for SiamRPN++ is different from the standard ResNet-50. The stride of the last two layers (Conv4 and Conv5) is reduced from 16 pixels and 32 pixels down to 8 pixels because a stride of 16 pixels or 32 pixels would not be suitable for Siamese object tracking [1]. Another major difference is the missing global average pooling layer ("Avg Pool") and the fully-connected layer ("FC"), depicted in Figure 3.11.

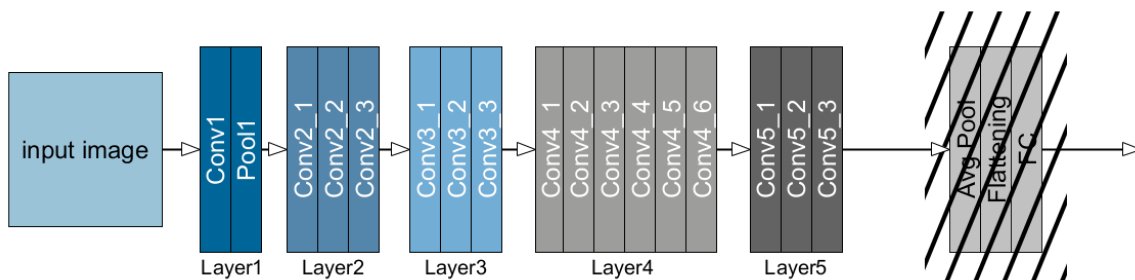


Figure 3.11: Architecture of the SiamRPN++ ResNet-50 [1]. The crossed-out area indicates the missing modules compared to the standard ResNet-50.

The authors from SiamRPN++ pre-train their ResNet-50 backbone on ImageNet [10] for image labeling. This network is also available in the PySOT repository, but it does not contain the final layers (global average pooling and fully connected) to generate class scores. But as described below, these final layers are necessary for the pruning process.

3.6.2 Pruning workflow

First, the global average pooling layer and the fully-connected layer are added at the end of the SiamRPN++ ResNet-50 that is provided in the PySOT repository. These layers are necessary to generate the classification scores required for calculating training and validation accuracies. In contrast to the layers from the ResNet-50 backbone, the added layers are only initialized and therefore need to be trained. The parameters of the ResNet-50 are already trained and therefore get fixed for this training process - only the newly added layers are actually trained. The idea is to take advantage of the already good weights from the pre-trained backbone and use it as a teacher for the added layers. This can be seen as a form of transfer learning, however, the difference is that the added layers are only initialized and not already trained on a dataset.

After testing several training parameter configurations, the parameters were set as follows:

- Batch size: 768

- Epochs: 90
- Learning rate schedule: Cosine Annealing without restart and initial learning rate 0.1

The expectation was to train the newly added layers relatively fast, within a few epochs. However, it has been observed that the maximum validation accuracy is only attained towards the end of the 90 epochs. The resulting ResNet-50 network achieved a Top-1 accuracy of 69.8% and a Top-5 accuracy of 89.5%. For the Top-1 accuracy, the predicted class with the highest probability is compared with the ground truth target class. The top 5 accuracy checks that the ground truth class is within the top 5 predicted classes.

Once the ResNet-50 of SiamRPN++ is extended with the required layers, pruning can be initiated. As described in the theory chapter (section 2.3.1), the impact of the convolutional channels on the accuracy is examined. Therefore each residual block (Figure 2.14) is equipped with three so-called index masks. After each convolution inside the residual block, an index mask is placed and initially filled with ones. If a convolution, for example, has 64 output channels, the corresponding index mask is filled with 64 ones. After the convolution, the result is multiplied by the index mask. The impact of a single channel is determined by setting the corresponding position in the index mask to 0 and calculating the accuracy. This allows for determining how the removal of this channel affects the accuracy. The decision of whether this channel is removable or not is based on the KL-divergence. A score of 0 indicates that the channel is redundant and therefore pruning this channel has little effect on the overall accuracy. The scores of all channels are saved for the next step: determining the threshold.

All scores are loaded into a single list and sorted in descending order. The top k channels get removed, resulting in a pruned and smaller network. If i describes the length of this list and thus the number of all channels, then the channels with index $k \in (k, i]$ will be removed. The compression rate is obtained by calculating the ratio of the number of parameters in the pruned network to the number of parameters in the original network. A minimum compression rate threshold (e.g., 0.3) is set for each layer to avoid any potential extreme issues such as having too few filters left in a layer after pruning. The parameter k is determined based on the target number of parameters that the pruned network should have. To achieve a certain number of parameters for the pruned network, it is necessary to test different values of k .

Once the channels to be removed have been identified, the most time and resource consuming process can be started: finetuning. The process of finetuning in pruning involves re-training the pruned network on the same dataset that was used to train the original network. The aim of finetuning is to regain the lost accuracy by adjusting the remaining parameters to compensate for those that were pruned. During the initial phase of finetuning, the network's weights (extended ResNet-50 of SiamRPN++) are loaded, and subsequently, a PyTorch model is created based on the previously generated score list. The

model is then loaded with the weights of the network, excluding the pruned ones. Subsequently, the actual training phase commences. In essence, there is no significant difference between training a standard ResNet from scratch and training the pruned model. All pruned networks are finetuned with the following parameters:

- Batch size: 256
- Epochs: 100
- Learning rate schedule: Cosine Annealing with 5 warmup epochs, no restart and initial learning rate 0.1

Once the process of finetuning the pruned network is finished, the layers that were added at the beginning can be removed. The result is a pruned SiamRPN++ ResNet-50 - ready to replace the original backbone. However, merely substituting the layers is insufficient due to the changes in weights caused by pruning. It is not assured that the remaining part of the SiamRPN++ network would be compatible with the new weights or produce results comparable to those of the original backbone. Hence a training run of the entire SiamRPN++ network is necessary (section 3.1.1).

Chapter 4

Experiments and Results

The following chapter presents the results of the conducted experiments, which build upon the methods introduced in the previous chapter 3. The experiments describe the accuracy and EAO of SiamRPN++ trackers with different TensorRT precisions and pruned backbones. Finally, the power and energy consumption of these trackers is determined.

4.1 Baseline model

In order to measure the effectiveness of any optimization, it is essential to have a baseline model for comparison. This baseline model represents the current level of accuracy or effectiveness that can be achieved with existing methods. Without a baseline, it would be difficult to determine whether a new approach is indeed an improvement over existing methods or simply performs similarly to or worse than the baseline. By comparing the performance of a new model to the baseline, it is possible to determine whether the new approach is worth pursuing further, or whether it is unlikely to provide significant performance improvements.

As mentioned in chapter 3, a Python implementation of SiamRPN++ is provided in the PySOT repository ¹. This repository contains a test script, which loads a given testing dataset and performs visual object tracking. Running this test script with the provided tracking network (including a ResNet-50 backbone) serves as the baseline for the following experiments.

Framework	Hardware	Accuracy	Robustness	EAO	FPS
PyTorch	NVIDIA Titan Xp	0.6	0.234	0.414	35
PyTorch	NVIDIA Jetson Xavier AGX	0.6	0.243	0.413	11

Table 4.1: Baseline model results: The first row are the results presented in the SiamRPN++ paper [1] and serve as a benchmark (except FPS).

¹<https://github.com/STVIR/pysot>

The first row in table 4.1 lists the test results published in the SiamRPN++ paper [1]. These results are achieved on the NVIDIA Titan Xp, a high-end GPU based on the Pascal architecture. Equipped with 3840 CUDA cores, the NVIDIA Titan Xp achieves up to 12 TFLOPS in single precision mode (FP32). As different GPU architectures are used, a direct comparison between Titan (Pascal) and Xavier (Volta) is difficult. Moreover, both GPUs have been designed for different application areas. According to NVIDIA, the Xavier AGX can deliver up to 11 TFLOPS in FP16 mode - but no value can be found for FP32 mode. Based on the number of available CUDA cores, it is plausible that the Titan is more powerful and achieves significantly more FPS than the Xavier AGX (table 4.1).

The performance measures between the two GPUs are expected to be identical as both used FP32 precision. However, there is a slight difference in robustness, with the Xavier performing 0.01 percentage points worse. Despite this, no differences were observed in the EAO score or accuracy. Regarding the total number of images in the test dataset, this means that the tracker lost the target object one more time in total. However, this does not necessarily mean that both trackers lost the target object at the same locations and that the tracker on Xavier lost it one more time additionally. This is because the exact position at which the tracker lost the target object is not relevant for the calculation of robustness. On the other side, it is important for the calculation of the EAO score. As mentioned in section 3.3.1, the EAO measure is calculated by averaging the EAO curve over the typical short-term sequence lengths interval (Figure 3.6 and 3.7). If the tracker on Xavier loses the target in a way that the expected average overlap curve is not affected in this interval, then the resulting EAO score is the same. When only robustness is considered, it has an impact, since overall the target object was lost one more time and the tracker on Xavier is therefore less robust.

4.2 TensorRT engines

The process for evaluating the TensorRT engines relies on the test script that is also utilized to evaluate the PyTorch models (section 4.1). However, instead of loading the PyTorch model, TensorRT engines are created or loaded (deserialized) if they already exist. Prior to creating these engines, the ONNX models must be generated from the original PyTorch model. This is done via an additional script. Following this step, the tracker is fed with each frame as the script iterates over all videos, like in the original test script.

As already mentioned in section 3.5, TensorRT supports three precision modes: 32 bit floating-point (FP32), 16-bit floating-point (FP16) and 8-bit integer (INT8). These three precision modes are evaluated in the following.

Using FP32 precision on the Xavier AGX can result in more accurate models, but it may also come at

the cost of reduced performance compared to using lower precision options like FP16. The expectation is that the TensorRT engines with FP32 precision maintain the performance measures of accuracy, robustness and EAO. But the inference latency is expected to be significantly smaller, resulting in a higher FPS rate.

Framework	Precision	Accuracy	Robustness	EAO	FPS
TensorRT	FP32	0.6	0.239	0.392	13.1
TensorRT	FP16	0.6	0.248	0.389	49.9
TensorRT	INT8	0.6	0.281	0.347	65.7

Table 4.2: Tracking results on VOT2018 for TensorRT engines

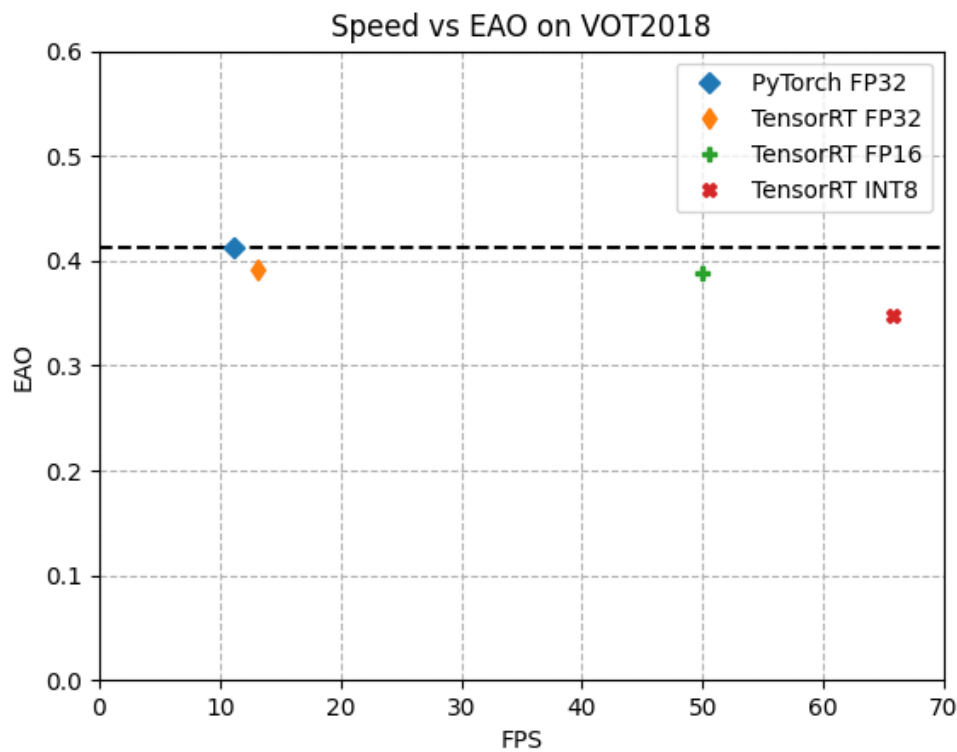


Figure 4.1: Comparison of quality (EAO) and speed of the baseline tracker (PyTorch FP32) and the tracker with TensorRT engines.

Figure 4.1 illustrates a comparison of speed (in FPS) versus EAO of the three TensorRT trackers (table 4.2), along with the result of the baseline PyTorch tracker (dashed line). The TensorRT FP32 tracker shows improvement regarding speed, but there is still a significant gap to real-time speed (30 FPS). The quality (EAO) of the tracker is below that of the baseline tracker, which is contrary to what can be expected as the baseline tracker is also executed in FP32 precision. Surprisingly, the robustness of the TensorRT FP32 is superior to that of the baseline tracker. The reason why the robustness is better, but the EAO is worse, is again due to the way in which the EAO is calculated (section 4.1). During the

typical sequence length (Figure 3.7), the TensorRT FP32 tracker's EAO curve must be lower, resulting in a lower EAO score. The overall accuracy remains at 0.6. Thus, the EAO curve for the TensorRT FP32 tracker must be higher for sequence lengths N_S outside the typical sequence length interval.

The TensorRT FP16 tracker operates at almost 4 times (3.8) the speed of the TensorRT FP32 tracker while maintaining accuracy and EAO. It exceeds speed constraints for real-time applications by far (49.9 FPS), making it an attractive option. But the robustness is slightly lower compared to the TensorRT FP32. The number of total lost target objects is increased by two. Nevertheless, I consider the tracking quality of the TensorRT FP16 as good as the TensorRT FP32 tracker, while performing well beyond real-time speed. In summary, I would definitely prefer the TensorRT FP16 tracker over the TensorRT FP32 tracker.

Operating the TensorRT tracker with 8-bit integer precision (INT8) enables another speedup compared to the TensorRT FP 32 tracker. The TensorRT INT8 tracker achieves 65.7 FPS, which is 5 times as fast as the TensorRT FP32 tracker. Compared to the TensorRT FP16 tracker, it is roughly one-third faster. Unfortunately, but as expected, the TensorRT INT8 tracker can not keep up with the tracking quality of the other trackers. Table 4.2 shows a dropoff in the EAO score: -11.5% related to the TensorRT FP32 tracker. Also, the robustness is significantly worse: $+16.6\%$ related to the TensorRT FP32 tracker or 9 additional losses of the target object. Surprisingly the TensorRT INT8 tracker is able to maintain accuracy. But this effect comes from the reset-based short-term tracking protocol mentioned in section 3.3.1. Due to the increased number of target losses, the tracker is reset and initialized more often. In a way, this gives the tracker a starting advantage, because it is allowed to start again at the highest possible accuracy. So in total, I consider the loss in tracking quality (EAO and robustness) more important than the increase in FPS, therefore I would still prefer the TensorRT FP16 tracker.

NVIDIA provides the TensorRT Engine Inspector, which allows inspecting TensorRT engines. It is possible to view detailed information, such as layer-type average latencies.

Figures 4.2, 4.3 and 4.4 illustrate the layer-type average latencies for the target, search and xcorr engine respectively. They show that convolutions play the most important role for the target and the search engine. For the target and the search engine, the other layer types (Reformat and Pooling) make only an insignificant contribution to the total latency. The reformat layer is used for data format conversion between precisions and different tensor data formats, such as NHWC, NCHW, or CHW. N describes the number of images in the batch (for tracking always equal 1), H and W for the height and width of the image and C denotes the number of channels. FP16 significantly reduces the average convolution latency, especially for the search engine (by a factor of 5.5).

In the figure for the xcorr engine (Figure 4.4) an additional type of layer is displayed. Myelin is an

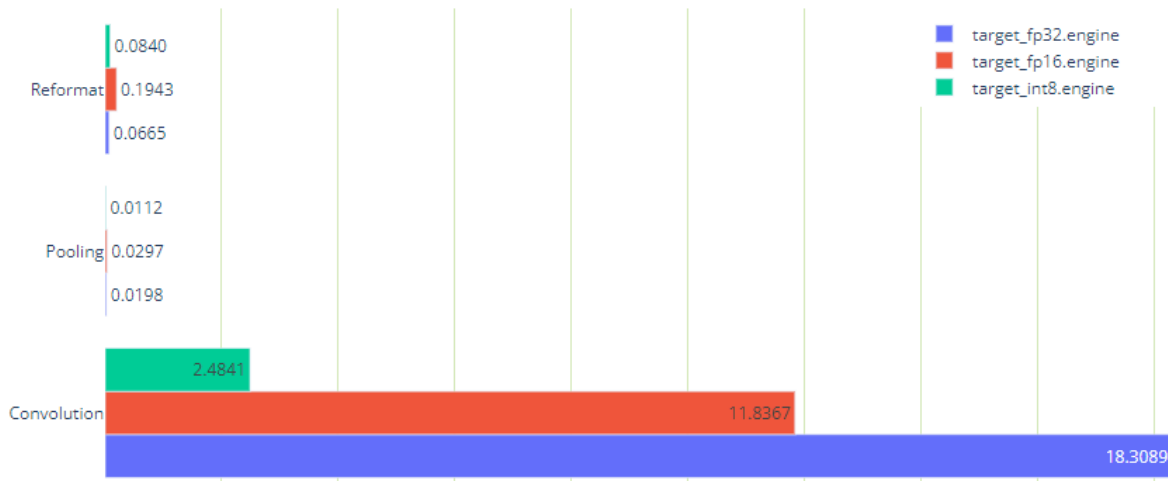


Figure 4.2: Layer-type latencies (average in ms) of the target engine for FP32, FP16 and INT8.

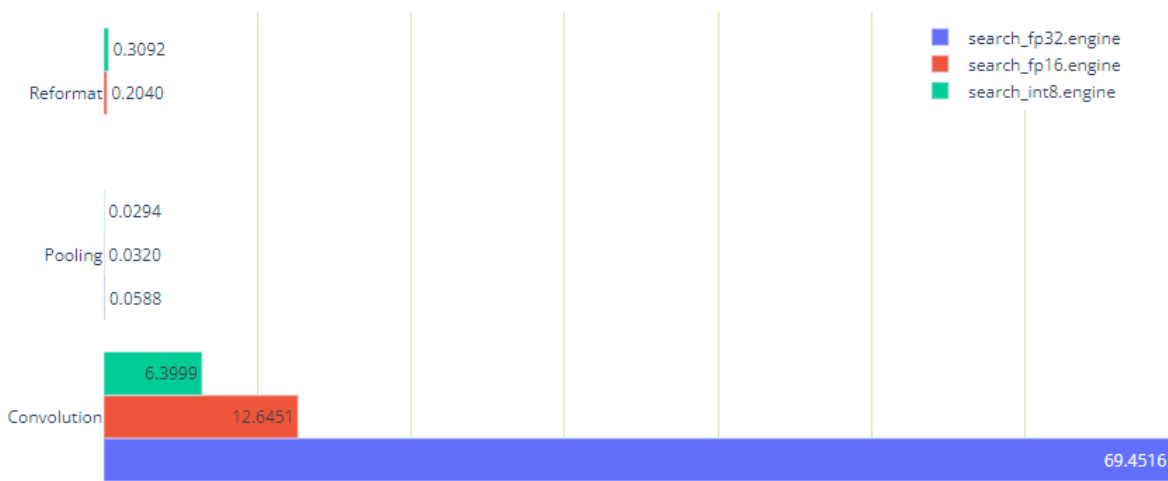


Figure 4.3: Layer-type latencies (average in ms) of the search engine for FP32, FP16 and INT8.

internal component of TensorRT and NVIDIA currently does not plan to reveal any details or publish documentation. According to the engine graph, the Myelin layer is used to split or slice the input tensor. But because of the missing documentation, no further analysis is possible.

Another interesting point is the average latencies for the reformat layer. When using INT8 precision, the average latency nearly doubles compared to FP16. And in total, the xcorr INT8 does not have a lower latency than the FP16 engine. According to the engine graph, the input tensor of the FP16 engine is reformatted at once, while for the INT8 engine reformatting is placed after the Myelin layer for each sliced tensor.

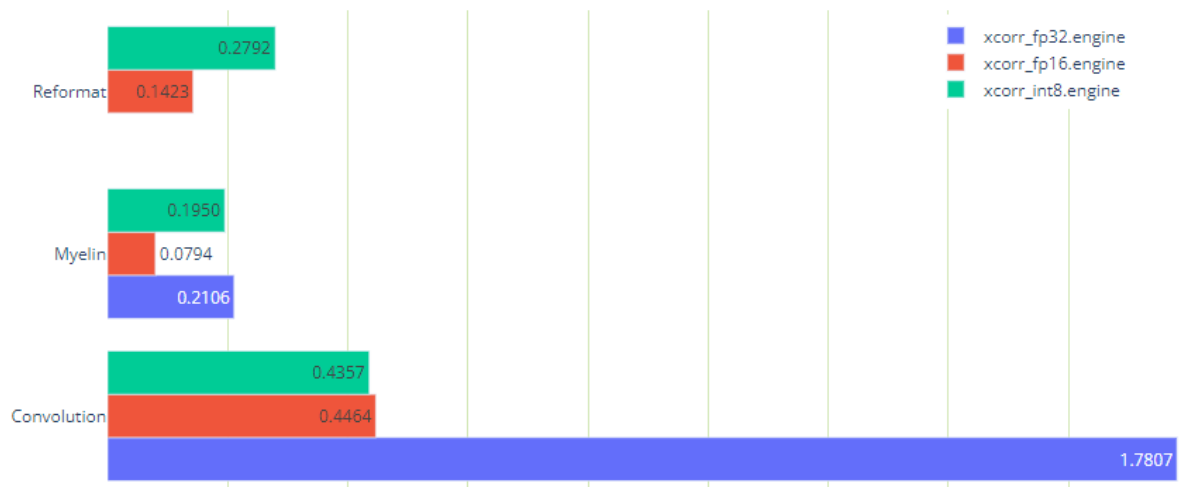


Figure 4.4: Layer-type latencies (average in ms) of the xcorr engine for FP32, FP16 and INT8.

As section 3.5 mentions, operating the TensorRT tracker in INT8 precision requires calibration with real input data. Therefore, a representative calibration dataset has to be extracted from the actual testing dataset. Because three TensorRT engines are used, three separate calibration datasets have to be created. The methodology for the target and the search engine is the same. First, the desired total number of images has to be specified. For the engines of the TensorRT INT8 tracker of table 4.2, 1000 images were specified. The testing dataset (VOT2018) consists of 60 videos of unequal length. In order not to prefer longer videos, the same number of images per video is used. In the example of 1000 desired images in total, that would be 23 images per video (rounded up). The actual selection of images in a video is random-based. So 23 random indexes between 0 and the length of a video are used to select the images for calibration.

The process of creating the calibration dataset for the xcorr engine is different. The xcorr engine does not get actual image input data. Instead, the outputs of the search engine are used as inputs for the xcorr engine. The outputs of the target engine are used to refit the xcorr engine, but this does not matter for the INT8 calibration as it is no actual input. The methodology for creating the calibration

dataset of the xcorr engine is as follows. In order to get representative outputs of the search engine, a TensorRT FP32 tracker is created. Then this tracker tracks each video of the test dataset. Again, at 23 random indexes between 0 and the length of the video, the outputs of the search engine are saved.

Actually, with this process of generating the calibration dataset(s), the TensorRT engines get biased. Because the calibration dataset consists of images that also appear in the testing dataset. So each engine already has seen a small part of the testing dataset. This gives the TensorRT INT8 tracker a starting advantage compared to the other trackers. But this bias can not be avoided, because unlike in object detection datasets, images used for calibration can not simply be removed from the testing dataset. Removing the used images from the testing dataset may have a significant impact because the images are related to each other. For real applications, calibrating the TensorRT INT8 engines will be harder due to the fact, that the real input data is not known and the tracker will not have seen this data before. It is therefore to be expected that the tracking quality of the TensorRT INT8 tracker will be lower for real applications.

When comparing different trackers, the notion of a "best" tracker varies with the tracker application [73]. For applications like sports analytics, the tracking quality of the estimated player position is crucial for analyzing player accelerations and velocity, while real-time performance is not necessary. Additionally, losing the target object is not critical as tracker reinitialization by the user is allowed at any point. In contrast, there are applications like autonomous driving, where tracking autonomy is crucial. A more robust tracker would be preferred over an accurate but non-robust tracker.

All videos of the VOT2018 testing dataset are per-frame annotated with the following visual attributes: occlusion, size change, motion change, illumination change and camera motion. Figure 4.5 illustrates the performance of the baseline tracker and the three TensorRT trackers.

Regarding to the VOT2017 challenge [46], where the same testing dataset is used, the most challenging attribute for both robustness and accuracy is occlusion. The results for occlusion (Figure 4.5) show that the ranking of the trackers is the same as in Figure 4.1. When it comes to changes in object size, the TensorRT INT8 tracker faces more difficulties compared to the other trackers, which seem to handle this task equally well. Accuracy is important for motion change [46], so the baseline PyTorch FP32 tracker performs best. Unexpectedly, the TensorRT INT8 tracker performs better on motion change than the TensorRT FP16 tracker. To the best of my knowledge, this comes again from the EAO calculation scheme. Table 4.2 shows that the TensorRT INT8 tracker loses the target more often (lower robustness). Due to the reset-based short-term tracking protocol, the sequences used for calculating the EAO score are different than the sequences used when calculating the EAO score for the TensorRT FP16 tracker. Regarding illumination change, there is no significant difference for the Ten-

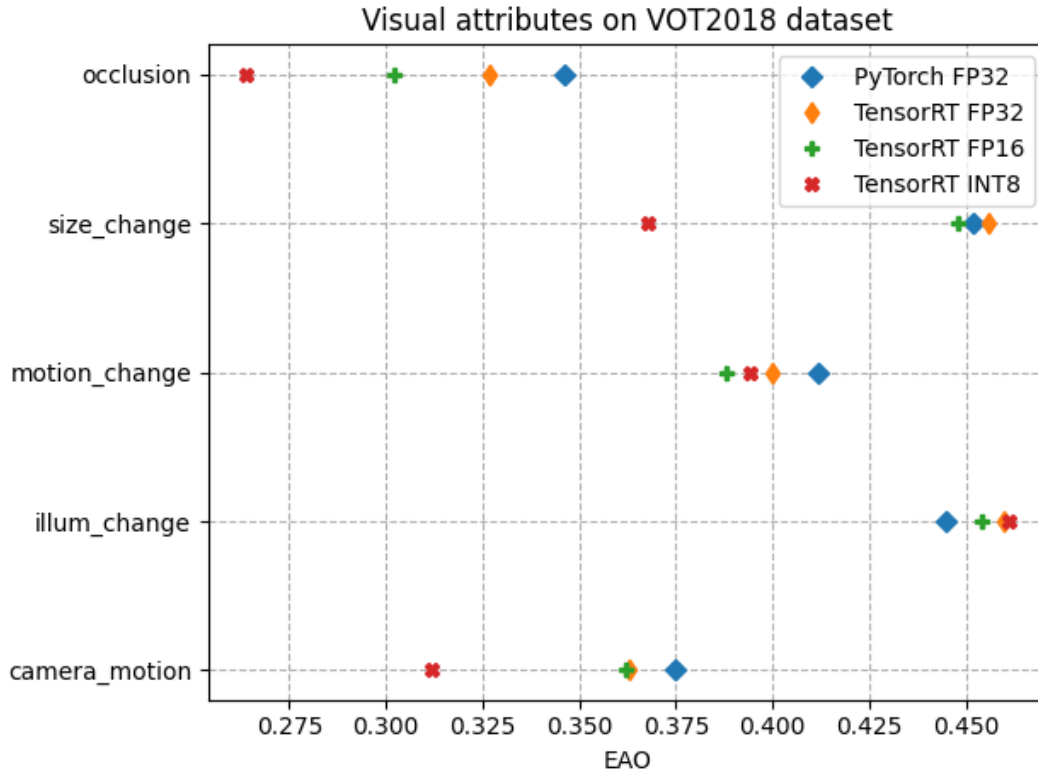


Figure 4.5: Evaluation with respect to visual attributes on the VOT2018 testing dataset.

TensorRT trackers, while the baseline PyTorch FP32 tracker performs slightly worse. For camera motion, the distribution is again the same as for the overall testing dataset (Figure 4.1). As already seen in the size change, the TensorRT INT8 tracker shows a significant gap to the other trackers.

TensorRT is not fully open-source, the core components are proprietary and closed-source. So it is necessary to rely on NVIDIA to fulfill the specified functionality. When evaluating the TensorRT trackers on the NVIDIA Xavier with TensorRT version 7.1.3.0, the results were inexplicably false and the trackers were not able to actually track an object. While searching for the cause of these failures in my own implementation, the solution to this problem was updating TensorRT to version 8.0.3.4. So apparently TensorRT 7.1.3.0 included a bug that caused the trackers to report wrong results.

4.2.1 Mixing precisions

Separating the baseline model into three networks (Section 3.4 and Figure 3.9) allows to mix the precisions for the TensorRT engines. The target engine computes the weights used to refit the xcorr engine. This is done at initialization - ideally only once per video. Even when multiple re-initializations are necessary, the target engine needs to be run significantly less often compared to the other two engines.

A significant increase in speed (FPS) is therefore not to be expected. On the other side, the output of the target engine should be as accurate as possible for refitting the xcorr engine. Therefore, the target engine with FP32 precision should produce a tracker with greater tracking quality than trackers with lower target engine precision.

The search engine is responsible for computing the features of every single frame after initialization. The quality of these features is crucial for producing accurate and robust tracking results. Therefore the search engine with FP32 precision is expected to produce the tracker with the best tracking quality. On the other side, as the search engine is executed on every frame, it has the greatest impact on the latency of the tracker. So reducing the precision for this engine should significantly speed up the tracker's throughput (FPS).

Due to its comparatively small number of layers, the xcorr engine is not expected to significantly impact the tracker's latency. Its function involves processing the outputs of the target and the search networks, meaning the quality of its results is partially dependent on their quality. Nevertheless, the precision of the xcorr engine itself is also expected to influence the quality of the tracker.

Target	Search	Xcorr	Accuracy	Robustness	EAO	FPS
FP32	FP32	FP16	0.605	0.276	0.367	12.9
FP32	FP32	INT8	0.539	0.754	0.170	13.4
FP16	FP32	FP32	0.602	0.267	0.394	12.8
INT8	FP32	FP32	0.605	0.272	0.377	13.2
FP16	FP32	FP16	0.602	0.258	0.395	12.8
FP16	FP32	INT8	0.538	0.773	0.168	13.8
INT8	FP32	FP16	0.602	0.248	0.398	13.3
INT8	FP32	INT8	0.534	0.679	0.183	13.6
FP32	FP16	FP32	0.605	0.253	0.389	33.6
FP32	FP16	FP16	0.603	0.248	0.388	41.8
FP32	FP16	INT8	0.536	0.745	0.168	42.7
FP16	FP16	FP32	0.602	0.262	0.396	41.6
INT8	FP16	FP32	0.598	0.267	0.366	50.1
FP16	FP16	INT8	0.539	0.862	0.151	42.5
INT8	FP16	FP16	0.603	0.258	0.387	52.8
INT8	FP16	INT8	0.534	0.684	0.181	52.9
FP32	INT8	FP32	0.604	0.272	0.365	43.0
FP32	INT8	FP16	0.602	0.253	0.371	59.1
FP32	INT8	INT8	0.532	0.768	0.167	64.0
FP16	INT8	FP32	0.597	0.290	0.362	59.3
INT8	INT8	FP32	0.595	0.276	0.371	60.9
FP16	INT8	FP16	0.606	0.290	0.359	78.8
FP16	INT8	INT8	0.534	0.745	0.174	61.5
INT8	INT8	FP16	0.601	0.276	0.357	55.3

Table 4.3: Tracking results when mixing the engine's precisions.

Table 4.3 lists the results for mixing the precisions of the tracker engines. The results are sorted by

precision of the search engine. The EAO value of a tracker with an INT8 target engine is higher than expected, suggesting that calibration with later test data is more advantageous than initially thought. Even with an FP16 engine, the EAO value remains higher than that of an FP32 engine. Despite this, in practical applications, FP16 is still my preferred choice.

As expected, the precision of the search engine is the primary factor affecting the speed of the tracker. This is also highlighted by Figure 4.6, which shows the distribution of tracking speeds for the different precisions (FP32, FP16 and INT8) of the search engine. The impact of the precisions is clearly limited to specific ranges (FP32 < FP16 < INT8), whereas no such influence is observed in the other engines. Trackers with a search engine in FP32 precision perform well under the real-time threshold. The tracker using an FP16 search engine performs comparably to an FP16 tracker but with a significantly smaller median (42.6 vs 49.9). Also, the tracker with an INT8 search engine performs comparably to an INT8 tracker.

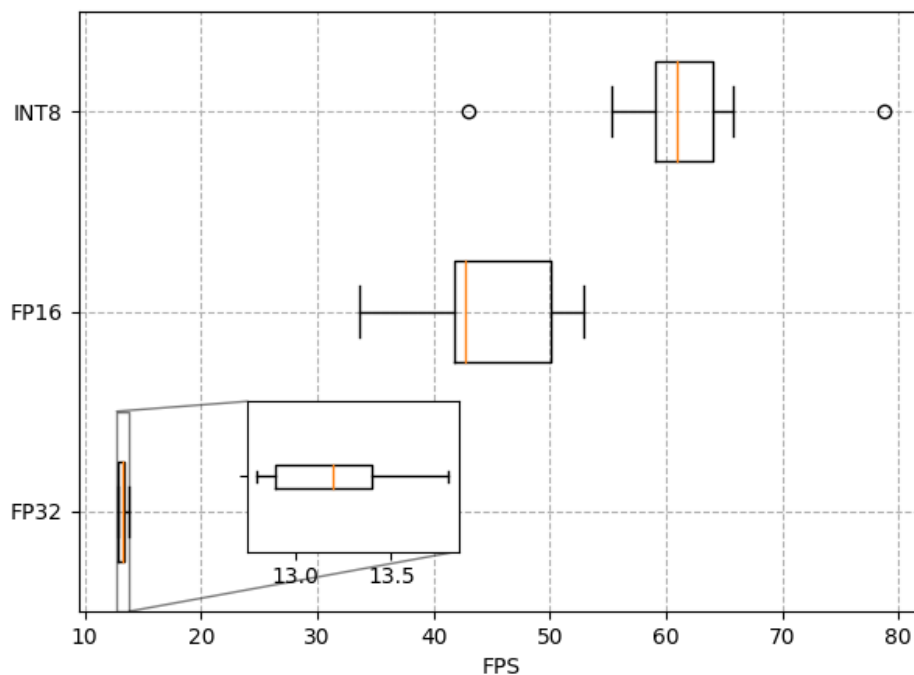


Figure 4.6: FPS results for the tracker with search engine in FP32, FP16 and INT8.

Trackers using an FP32 or FP16 xcorr engine perform similarly in terms of both EAO and FPS. However, there is a noticeable decrease in EAO with the INT8 xcorr engine, which may be due to calibration (Figure 4.7). The outlier in the plot represents the EAO value when all engines are operated in INT8 precision. It is important to note that the calibration scheme for the other INT8 xcorr engines

has not been changed.

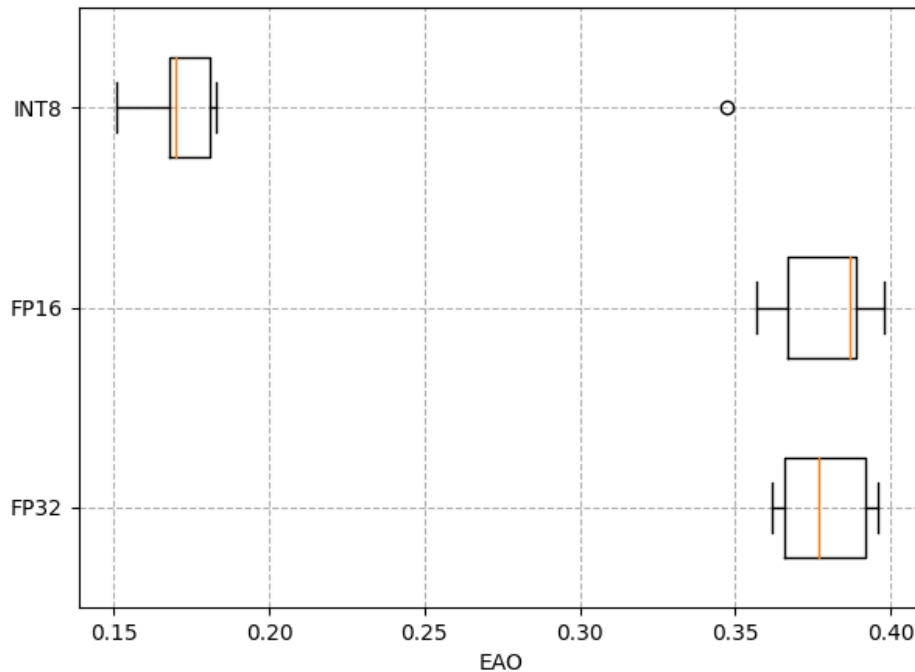


Figure 4.7: EAO results for the tracker with xcorr engine in FP32, FP16 and INT8.

In summary, a combination of different precisions can result in achieving the highest EAO (INT8 | FP32 | FP16 for 0.398) and FPS values (FP16 | INT8 | FP16 for 78.8), contrary to initial expectations. However, it is important to note that trackers using INT8 engines are calibrated using data from the test dataset, which may give them a slight advantage. As a result, I would prefer to use trackers without INT8 engines.

4.2.2 Pruned backbone

As described in the methodology in section 3.6, the first step in pruning the backbone of the PYSOT model is training the two new attached layers on the ImageNet classification dataset. This is necessary because the PYSOT model does not have the classification layer required for training such models on the ImageNet classification dataset. The training was performed with the parameter configuration mentioned in section 3.6 on the NVIDIA A100 GPU nodes of the VSC5 (section 3.2).

Figure 4.8 shows the Top-1 accuracy over the number of epochs for training the backbone of the PYSOT model on ImageNet. Top-1 accuracy means, that the predicted class from the network is actually the annotated ground truth class. Unfortunately, the best Top-1 accuracy is clearly below the results from the original ResNet50 from He et al. [4]. The authors of SiamRPN++ [1] did not provide

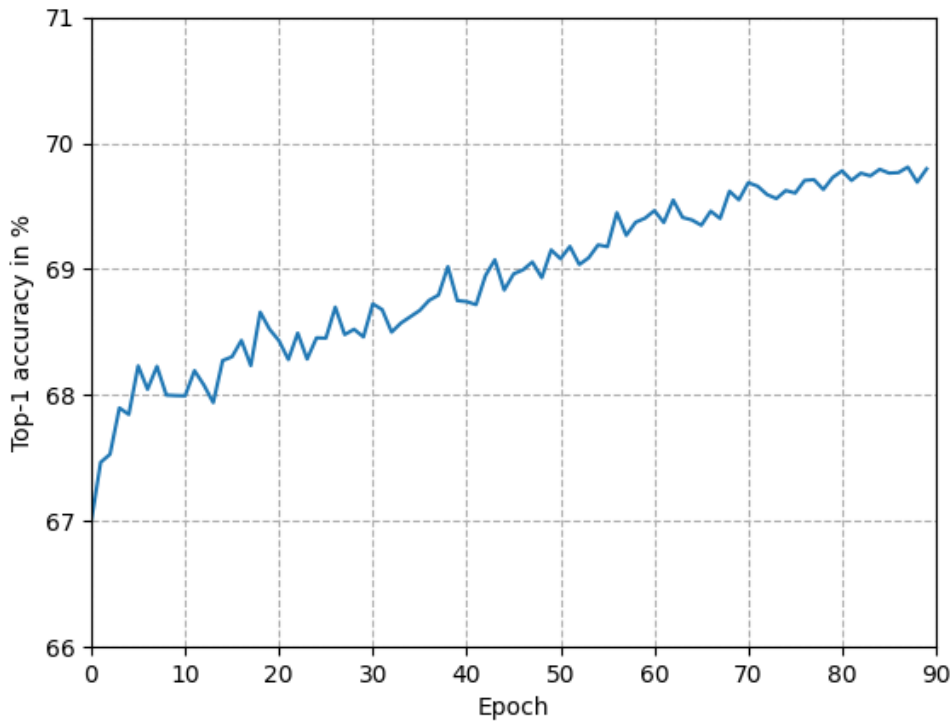


Figure 4.8: Top-1 accuracy for training the backbone on the ImageNet classification dataset with the two attached layers (details in section 3.6.2)

comprehensive information about the backbone training, and also did not publish a complete backbone with the classification layer. As a result, the achieved Top-1 accuracy is the best outcome that could be obtained under these circumstances.

The next step in the pruning workflow is to examine the impact of the convolutional channels on the accuracy. The result of this step is a list containing the score for each channel. Based on this list, the parameter k is determined. The baseline PYSOT model has $45.5M$ parameters. k is chosen in such a way that the resulting pruned networks have a number of parameters that is a multiple of one-fourth to one-third of the number of parameters in the original network (Table 4.4). Afterwards, the most time and resource-consuming process is started: finetuning. As described in section 3.6, finetuning is basically retraining the pruned network on the same dataset it was trained before. Table 4.4 lists the number of parameters and their corresponding finetuning results.

It can be observed that all pruned networks (rows 2 to 5 in table 4.4) outperform the original network (row 1 in table 4.4) in terms of accuracy - both Top-1 and Top-5 accuracy. This strengthens the assumption from before that the initial training of the network could be optimized, leaving room for improvement. Luo et al. [9], which proposed the used method CURL, achieve a Top-1 accuracy of 73.39% and a Top-5 accuracy of 91.46% for their pruned network with 6.67M parameters. He et

Parameters in M	Compression ratio	Top-1 Accuracy	Top-5 Accuracy
45.5	1	69.796	89.524
30	0.66	75.948	92.766
22.5	0.5	75.492	92.452
12	0.26	74.346	91.942

Table 4.4: Results for finetuning the pruned networks. *Note: The network with 45M parameters is the non-pruned network.*

al. [4] accomplish 77.15% Top-1 accuracy and 93.29% Top-5 accuracy with 25.56M parameters. Reducing the number of parameters also decreases the accuracy. However, this loss is acceptable since the pruned networks fall between the accuracy levels of these two reference networks. Figure 4.9 depicts the progress of Top-1 accuracy and Top-5 accuracy over the number of epochs. According to the graphs, all three pruned networks reach a Top-1 accuracy of 60% within the first 10 epochs of the fine-tuning process. Furthermore, they gradually approach their maximum accuracy towards the end of the finetuning.

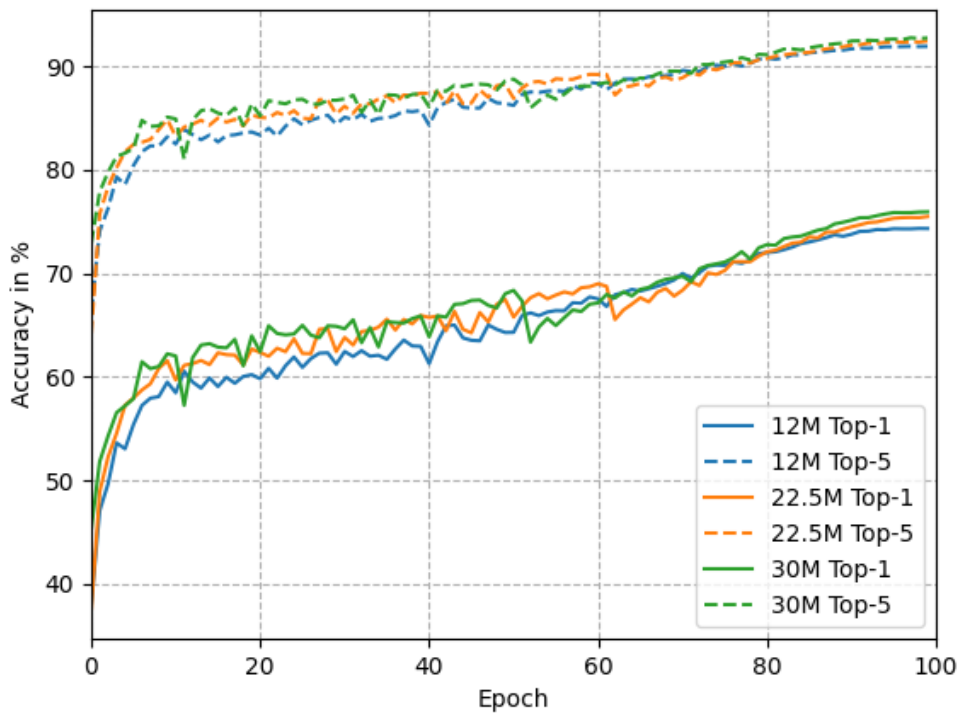


Figure 4.9: Visualization of the finetuning of the pruned networks from table 4.4

The graph for the network with 22.5M parameters in Figure 4.9 displays a significant drop in both accuracies at epoch 62, followed by a gradual recovery over the next few epochs. The same behavior is seen for the network with 30M parameters at epoch 52. The reason behind this drop is the time limit

for executing jobs on the NVIDIA A-100 nodes of the VSC5. As a result, the finetuning process had to be restarted for these two networks after exceeding the maximum time limit. Ideally, finetuning should have been resumed seamlessly, but the graphs indicate that it took several epochs to restore and further improve the accuracies.

When the backbone is pruned, the last step in the pruning workflow is training the entire tracker. The PYSOT repository offers a configuration for training the tracking network. This configuration specifies e.g. the type of the backbone, the number of epochs, which datasets the network shall be trained on, or parameters for data augmentation. The starting point for each tracker training is a pretrained backbone. Unfortunately, running the tracker training with the published configuration and published ResNet50 backbone, the results from the paper [1] can not be reproduced. It has to be mentioned that the optimal training parameters may vary depending on the training infrastructure. For example, the learning rate depends on the number of GPUs available, the batch size and also other training parameters. In the end, it took some trial and error to find the optimal training parameter settings.

4.2.3 TensorRT engines with pruned backbone

In the published training configuration, the ResNet50 backbone is left untrained for the first 10 epochs. Afterwards, it is trained together with the rest of the tracking network. The key modification is that during the tracker training, the ResNet50 backbone is excluded from the training process entirely.

Parameters in M	Framework	Precision	Accuracy	Robustness	EAO	FPS
45.5	PyTorch	FP32	0.572	0.267	0.344	11.2
45.5	TensorRT	FP32	0.564	0.300	0.328	13.5
45.5	TensorRT	FP16	0.549	0.286	0.323	51.6
45.5	TensorRT	INT8	0.535	0.459	0.240	62.1
30	PyTorch	FP32	0.519	0.576	0.216	16.0
30	TensorRT	FP32	0.478	0.637	0.190	19.6
30	TensorRT	FP16	0.478	0.693	0.183	52.0
30	TensorRT	INT8	0.459	0.801	0.164	66.9
22.5	PyTorch	FP32	0.538	0.534	0.239	19.3
22.5	TensorRT	FP32	0.524	0.562	0.217	24.0
22.5	TensorRT	FP16	0.514	0.585	0.210	57.9
22.5	TensorRT	INT8	0.466	0.796	0.166	73.8
12	PyTorch	FP32	0.518	0.613	0.212	26.6
12	TensorRT	FP32	0.496	0.651	0.190	35.7
12	TensorRT	FP16	0.503	0.656	0.188	59.3
12	TensorRT	INT8	0.476	0.848	0.159	94.4

Table 4.5: Tracking results on VOT2018 for networks with original number of parameters (45.5M) and pruned ResNet50 backbones (30M, 22.5M and 12M).

Table 4.5 lists the result of training a tracker with a backbone of the original size (45.5M parameters) and with pruned backbones of various degrees (30M, 22.5M and 12M parameters). Figure 4.10 provides a visualization of this table, plotting the speed over the EAO score.

It can be observed that the same pattern regarding speed is recognizable between the different precisions of each pruning level (compression ratio). The TensorRT INT8 tracker is always the fastest, followed by TensorRT FP16, TensorRT FP32, and PyTorch FP32. The smaller the compression ratio, the greater the speed. This was also expected based on the results from the previous sections. What also can be observed is that the TensorRT INT8 tracker of the network with 45.5M parameters is slower than the TensorRT INT8 tracker of the network with the original backbone from PYSOT. The number of parameters are exactly the same for both networks. I assume that TensorRT executed different optimizations for the tracker with the 45.5M parameter backbone due to a different weight distribution compared to the tracker with the original backbone from PYSOT. The different optimization strategy then results in lower execution speed.

Unfortunately, the EAO drops significantly when reducing the number of parameters or in other words, increasing the pruning compression. As mentioned before, there is already a significant drop of 0.07 points in EAO when training the tracker with a backbone of the original size (45.5M parameters) compared to the baseline tracker (EAO of 0.414). Although PYSOT released training configurations, it is not entirely clear how exactly this tracker was trained. Similar to speed, also a pattern for EAO can be observed. Within a pruning level, the PyTorch FP32 tracker has always the highest tracking quality. Subsequently, the TensorRT FP32 tracker follows at a notable margin, with the TensorRT FP16 tracker closely behind. As seen in the baseline results (table 4.2 and figure 4.1), a TensorRT FP16 tracker yields results of comparable quality to a TensorRT FP32 tracker but in significantly less time. However, when increasing the pruning compression, this gap narrows and a speedup of e.g. 4 (for the baseline tracker) is no longer present.

To summarize the results from table 4.5 and figure 4.10: As the training of a tracker is sensitive and the published training configuration does not yield the results from the paper [1], there is a significant gap in EAO already when training a tracker with a backbone of the original size (45.5M parameters). It can be said that the tracking quality of the trackers with pruned backbones is not satisfying. For the tracker with a 12M backbone, the EAO score is only half compared to the baseline tracker (table 4.2). On the other side, the results fall within the expected range in terms of speed. Increasing the pruning compression leads to an increase in speed.

However, I would prioritize the loss of EAO more than the gain in speed. As a result, I would prefer the TensorRT FP16 tracker with the pretrained backbone (table 4.2 and figure 4.1) over all other trackers.

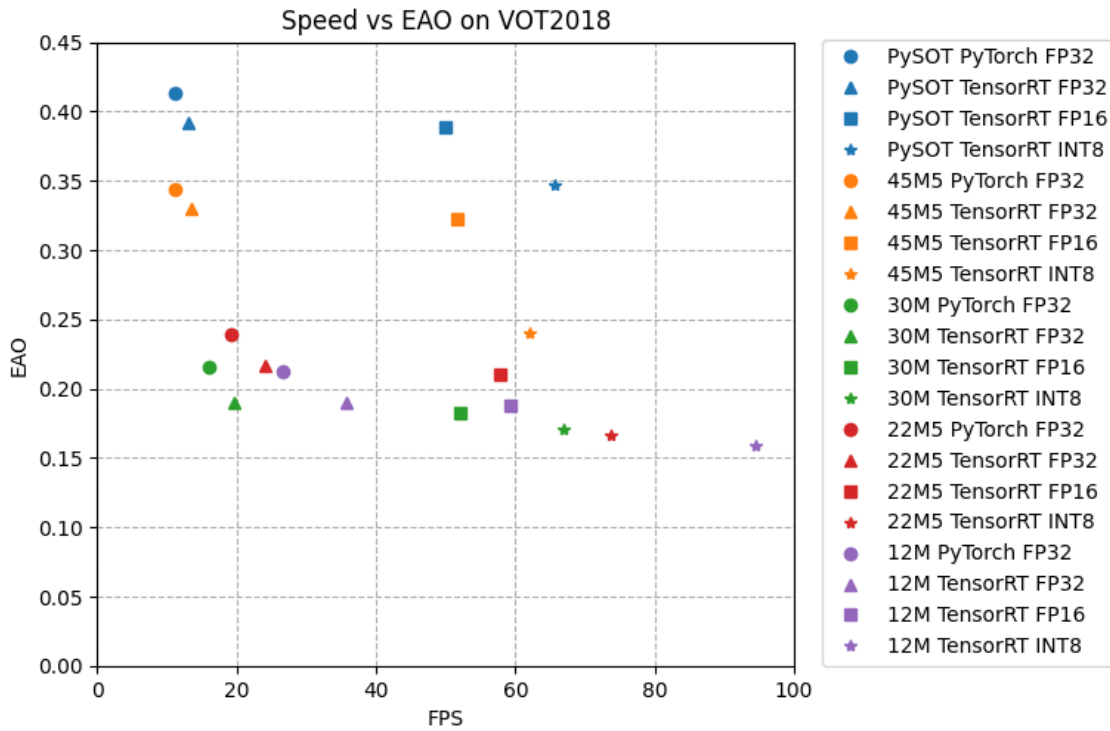


Figure 4.10: Comparison of quality (EAO) and speed of trackers with the original size backbone (45, 5M parameters) and pruned backbones (30M, 22.5M and 12M).

4.3 GPU Power and energy consumption

As mentioned in section 3.3.4, the NVIDIA Tegrastats tool is used to monitor the GPU performance in real-time during tracking application. The NVIDIA Xavier is run in the highest performance mode (MAXN) to ensure that the maximum computing power is provided. The Tegrastats statistics are sampled every 5 seconds, with the average value used as a base for computing power consumption. Running the tracker on the testing dataset generates a list of average power consumption values. The maximum value from this list is then identified as the power consumption. Together with the latency results (FPS) of a tracker, also the energy consumption of the GPU per frame can be calculated.

Framework	Precision	Power in W	Energy/frame in kJ
PyTorch	FP32	21.378	1.94
TensorRT	FP32	20.877	1.59
TensorRT	FP16	13.044	0.26
TensorRT	INT8	3.569	0.05

Table 4.6: GPU Power and energy consumption of the networks with the baseline ResNet50 backbones

Table 4.6 lists the maximum average power consumption (in W) for the tracker with the baseline

backbone and table 4.7 for the trackers with a pruned ResNet50 backbone (and the original size backbone). Figure 4.11 visualizes the results from table 4.7. The TensorRT FP16 tracker with the baseline ResNet50 backbone greatly reduces the power consumption by 37.52% and the energy consumption by 83.6% compared to the PyTorch FP32 tracker with the baseline ResNet50 backbone. Running this tracker in INT8 precision mode even reduces the power and energy consumption further by 82.9% and 96.59%. Unfortunately, this reduction in power consumption for the TensorRT INT8 tracker also comes with a substantial loss in tracking quality (EAO, table 4.2). Therefore, the TensorRT FP16 tracker remains the preferred choice, as the loss in EAO is negligible.

Parameters in M	Framework	Precision	Power in W		Energy/frame in J	
45.5	PyTorch	FP32	21.247	-	1.9	-
30	PyTorch	FP32	19.759	-7%	1.235	-35%
22.5	PyTorch	FP32	19.123	-9.99%	0.991	-47.84%
12	PyTorch	FP32	16.837	-20.76%	0.633	-66.68%
45.5	TensorRT	FP32	19.823	-	1.468	-
30	TensorRT	FP32	19.837	+0.07%	1.012	-31.06%
22.5	TensorRT	FP32	18.816	-5.08%	0.784	-46.59%
12	TensorRT	FP32	15.794	-20.32%	0.442	-69.89%
45.5	TensorRT	FP16	12.445	-	0.241	-
30	TensorRT	FP16	7.149	-42.56%	0.137	-43.15%
22.5	TensorRT	FP16	7.768	-37.58%	0.134	-44.4%
12	TensorRT	FP16	3.958	-68.2%	0.067	-72.2%
45.5	TensorRT	INT8	4.006	-	0.065	-
30	TensorRT	INT8	1.855	-53.69%	0.028	-56.92%
22.5	TensorRT	INT8	2.914	-27.26%	0.039	-40%
12	TensorRT	INT8	1.159	-71.07%	0.012	-81.54%

Table 4.7: GPU Power and energy consumption of the networks with pruned ResNet50 backbones. The percentage difference for each precision mode is related to the power consumption value of the 45.5M parameters backbone.

The trackers with a backbone of the original size (45.5M parameters) perform in the same range as the trackers with the baseline backbone. As the number of parameters and therefore the network size is identical, this can be expected. Reducing the number of parameters reduces the power consumed by the system. Except for the TensorRT FP32 tracker with the 30M parameter backbone, where the power consumption value is even slightly larger (19.837W) than the reference value (19.823W).

Similar to the observations made with speed, TensorRT INT8 provides the most optimal improvements in power and energy consumption, followed by TensorRT FP16, TensorRT FP32, and PyTorch FP32. TensorRT FP16 trackers significantly reduce power and energy consumption by up to 68.2% and 72.2%. However, the TensorRT FP16 tracker with a 12M parameter backbone has worse tracking quality than the reference tracker (45.5M parameter backbone). Nevertheless, the TensorRT FP16 tracker with a

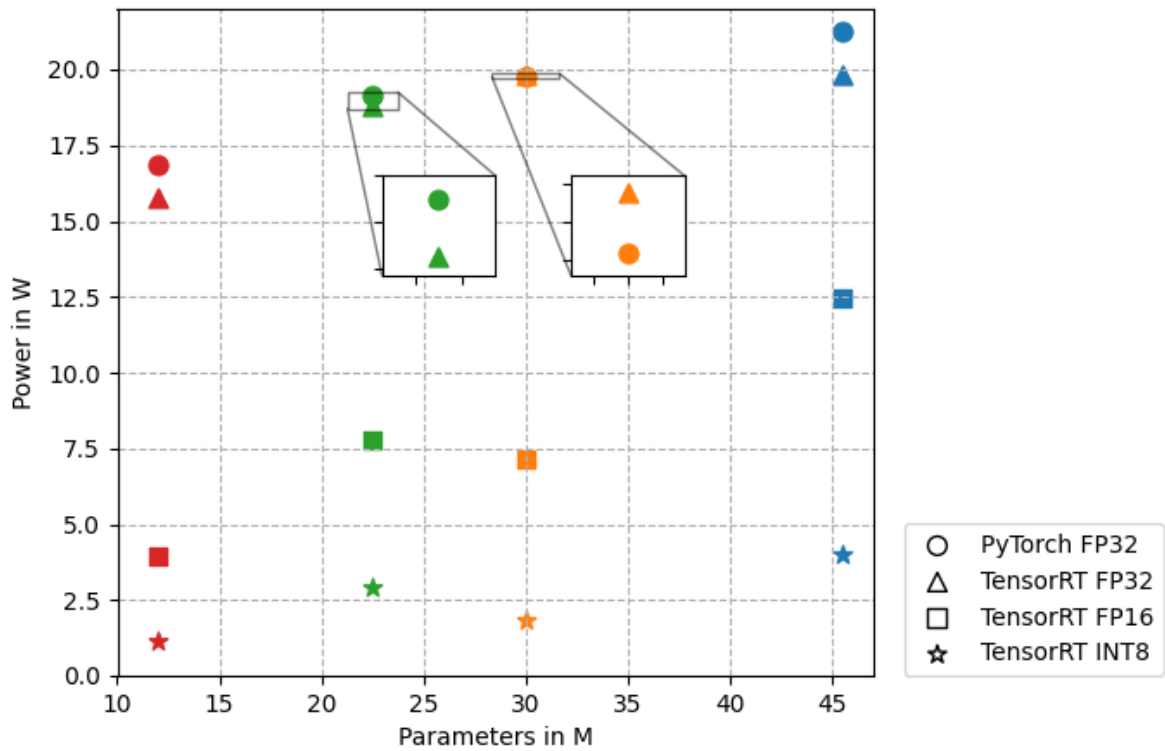


Figure 4.11: Visualization of the GPU power consumption for the networks with pruned ResNet50 backbones.

12M parameter backbone achieves a significant reduction in power and energy consumption compared to the TensorRT FP16 tracker with a 30M parameter backbone, while the tracking quality is even better.

Nevertheless, considering the loss in tracking quality to be more significant, I would prefer to use the TensorRT FP16 tracker with the pretrained backbone instead of the other trackers with a pruned backbone.

Chapter 5

Conclusion and future work

5.1 Conclusion

The goal of this thesis was to select a state-of-the-art Siamese object tracking network, optimize it for a specific embedded hardware platform and analyze the impact of different Convolutional Neural Network (CNN) optimization techniques. The tracker from the SiamRPN++ paper [1] was optimized for the NVIDIA Xavier AGX to enable real-time applications. According to the results, optimizing the tracking model with the NVIDIA TensorRT library and running the tracker with 16-bit floating point (FP16) precision yields the best overall performing tracker on the NVIDIA Xavier AGX. This tracker enables performance well beyond the real-time threshold while maintaining tracking quality (EAO, accuracy, and robustness). Compared to the baseline SiamRPN++ tracker running in the PyTorch Framework, a TensorRT FP16 tracker is able to run at almost 4 times the speed (49.9 FPS vs 11 FPS) and reduce power and energy consumption by 39% and 86.6%. In contrast, this tracker loses 0.025 in EAO compared to the PyTorch baseline tracker.

Running a tracker optimized with TensorRT in 32-bit floating point precision (FP32) does not significantly increase the tracking speed compared to the baseline PyTorch tracker. A tracker optimized with TensorRT in 8-bit integer (INT8) precision is able to run even faster than a TensorRT FP16 tracker, but the loss in tracking quality (EAO) is significantly high. Additionally, INT8 requires calibration of the TensorRT engines.

Since the baseline model needs to be separated into three models in order to run with TensorRT, mixing the TensorRT precisions was evaluated. But the result is that no combination of precisions exceeds the TensorRT FP16 tracker in tracking quality.

Optimizing the tracker by pruning the backbone proved to be challenging. The original backbone requires additional layers to enable pruning. These additional layers need to be trained. Second, train-

ing the entire tracker was also more difficult than anticipated. Running the training with the published training configuration led to significantly worse results than reported in the paper. It took considerable effort to find a training configuration where the loss in tracking quality was acceptable. As expected, trackers with pruned backbones show a significant increase in speed. However, the loss in tracking quality outweighs this speed improvement by a large margin. Therefore also here the recommendation for the TensorRT FP16 tracker remains valid.

The power and energy consumption results show that significant savings are possible by pruning the tracker backbones. A tracker with a 12M parameter backbone (25% of the parameters of the original backbone) requires 68.2% less power and 72.2% less energy in TensorRT FP16 precision.

This work shows that it is possible to execute a siamese object tracker (SiamRPN++) on a resource constrained hardware (NVIDIA Xavier AGX) at real-time speed while maintaining tracking quality.

5.2 Future work

This work showed that the NVIDIA TensorRT library can greatly utilize the power of the NVIDIA Xavier AGX and can speed up the SiamRPN++ tracker well beyond real-time. A topic to work on is the training of the SiamRPN++ ResNet50 backbone with the additional layers needed for classification. If improvements can be achieved here, it could also lead to better pruning results. Because if the original SiamRPN++ ResNet50 backbone shows higher accuracy, it would be expected that the accuracy after pruning is also higher.

Another issue to work on is the training of the overall tracker since there is a significant drop of 0.07 points (−17%) in EAO when training the tracker with the original backbone published by the SiamRPN++ authors. Along with potential improvements in pruning the backbone, an enhancement here can also result in overall better tracking quality of trackers with pruned backbones.

Bibliography

- [1] B. Li, W. Wu, Q. Wang, F. Zhang, J. Xing, and J. Yan, “SiamRPN++: Evolution of Siamese Visual Tracking with Very Deep Networks,” *arXiv:1812.11703 [cs]*, Dec. 2018, arXiv: 1812.11703. [Online]. Available: <http://arxiv.org/abs/1812.11703>
- [2] S. Albawi, T. A. Mohammed, and S. Al-Zawi, “Understanding of a convolutional neural network,” in *2017 International Conference on Engineering and Technology (ICET)*, Aug. 2017, pp. 1–6.
- [3] Alex Krizhevsky, “CIFAR-10 and CIFAR-100 datasets.” [Online]. Available: <https://www.cs.toronto.edu/~kriz/cifar.html>
- [4] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” *arXiv:1512.03385 [cs]*, Dec. 2015, arXiv: 1512.03385. [Online]. Available: <http://arxiv.org/abs/1512.03385>
- [5] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” *arXiv:1506.01497 [cs]*, Jan. 2016, arXiv: 1506.01497. [Online]. Available: <http://arxiv.org/abs/1506.01497>
- [6] M. Kristan, A. Leonardis, and J. Matas, “The Sixth Visual Object Tracking VOT2018 Challenge Results,” in *Computer Vision – ECCV 2018 Workshops*, L. Leal-Taixé and S. Roth, Eds. Cham: Springer International Publishing, 2019, vol. 11129, pp. 3–53, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-030-11009-3_1
- [7] L. Bertinetto, J. Valmadre, J. F. Henriques, A. Vedaldi, and P. H. S. Torr, “d,” *arXiv:1606.09549 [cs]*, Sep. 2016, arXiv: 1606.09549. [Online]. Available: <http://arxiv.org/abs/1606.09549>
- [8] B. Li, J. Yan, W. Wu, Z. Zhu, and X. Hu, “High Performance Visual Tracking with Siamese Region Proposal Network,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. Salt Lake City, UT: IEEE, Jun. 2018, pp. 8971–8980. [Online]. Available: <https://ieeexplore.ieee.org/document/8579033/>

- [9] J.-H. Luo and J. Wu, “Neural Network Pruning With Residual-Connections and Limited-Data,” in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Seattle, WA, USA: IEEE, Jun. 2020, pp. 1455–1464. [Online]. Available: <https://ieeexplore.ieee.org/document/9157802/>
- [10] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, Dec. 2015. [Online]. Available: <https://doi.org/10.1007/s11263-015-0816-y>
- [11] K. Matej, L. Aleš, M. Jiri, M. Felsberg, Luka, F. Gustavo, V. Tomaš, N. Georg, and P. Roman, “The Visual Object Tracking VOT2015: Challenge and results,” 2015. [Online]. Available: https://data.votchallenge.net/vot2015/presentations/vot_2015_presentation.pdf
- [12] M. Kristan, J. Matas, A. Leonardis, M. Felsberg, Luka Cehovin, and G. Fernández, “The Visual Object Tracking VOT2015 Challenge Results,” in *Visual Object Tracking Workshop 2015 at ICCV2015*. Santiago, Chile: IEEE, Dec. 2015. [Online]. Available: <https://hal.archives-ouvertes.fr/hal-01336773>
- [13] N. Zmora, H. Wu, and J. Rodge, “Achieving FP32 Accuracy for INT8 Inference Using Quantization Aware Training with NVIDIA TensorRT,” Jul. 2021. [Online]. Available: <https://developer.nvidia.com/blog/achieving-fp32-accuracy-for-int8-inference-using-quantization-aware-training-with-tensorrt/>
- [14] P. Li, D. Wang, L. Wang, and H. Lu, “Deep visual tracking: Review and experimental comparison,” *Pattern Recognition*, vol. 76, pp. 323–338, Apr. 2018. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0031320317304612>
- [15] 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://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html>
- [16] K. Fukushima, “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position,” *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, Apr. 1980. [Online]. Available: <http://link.springer.com/10.1007/BF00344251>
- [17] Y. LeCun, L. Bottou, Y. Bengio, and P. Ha, “Gradient-Based Learning Applied to Document Recognition,” 1998.

- [18] Hecht-Nielsen, "Theory of the backpropagation neural network," in *International 1989 Joint Conference on Neural Networks*, 1989, pp. 593–605 vol.1.
- [19] D. Ghimire, D. Kil, and S.-h. Kim, "A Survey on Efficient Convolutional Neural Networks and Hardware Acceleration," *Electronics*, vol. 11, no. 6, p. 945, Jan. 2022, number: 6 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: <https://www.mdpi.com/2079-9292/11/6/945>
- [20] A. Khan, A. Sohail, U. Zahoor, and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks," *Artificial Intelligence Review*, vol. 53, no. 8, pp. 5455–5516, Dec. 2020. [Online]. Available: <https://link.springer.com/10.1007/s10462-020-09825-6>
- [21] Andrej Karpathy, "CS231n Convolutional Neural Networks for Visual Recognition." [Online]. Available: <https://cs231n.github.io/convolutional-networks/>
- [22] S. Sharma, S. Sharma, and A. Athaiya, "ACTIVATION FUNCTIONS IN NEURAL NETWORKS," *International Journal of Engineering Applied Sciences and Technology*, vol. 04, no. 12, pp. 310–316, May 2020. [Online]. Available: <https://www.ijeast.com/papers/310-316,Tesma412,IJEAST.pdf>
- [23] Andrej Karpathy, "CS231n Convolutional Neural Networks for Visual Recognition." [Online]. Available: <https://cs231n.github.io/>
- [24] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Mar. 2015, arXiv:1502.03167 [cs]. [Online]. Available: <http://arxiv.org/abs/1502.03167>
- [25] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, "How Does Batch Normalization Help Optimization?" Apr. 2019, arXiv:1805.11604 [cs, stat]. [Online]. Available: <http://arxiv.org/abs/1805.11604>
- [26] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*. Santiago, Chile: IEEE, Dec. 2015, pp. 1026–1034. [Online]. Available: <http://ieeexplore.ieee.org/document/7410480/>
- [27] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, Mar. 2010, pp. 249–256, iSSN: 1938-7228. [Online]. Available: <https://proceedings.mlr.press/v9/glorot10a.html>

- [28] S. K. Kumar, "On weight initialization in deep neural networks," May 2017, arXiv:1704.08863 [cs]. [Online]. Available: <http://arxiv.org/abs/1704.08863>
- [29] M. Hussain, J. Bird, and D. Faria, *A Study on CNN Transfer Learning for Image Classification*, Jun. 2018.
- [30] A. Dhillon and G. K. Verma, "Convolutional neural network: a review of models, methodologies and applications to object detection," *Progress in Artificial Intelligence*, vol. 9, no. 2, pp. 85–112, Jun. 2020. [Online]. Available: <https://doi.org/10.1007/s13748-019-00203-0>
- [31] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation."
- [32] R. Girshick, "Fast R-CNN," 2015, pp. 1440–1448. [Online]. Available: https://openaccess.thecvf.com/content_iccv_2015/html/Girshick_Fast_R-CNN_ICCV_2015_paper.html
- [33] 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.
- [34] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA: IEEE, Jun. 2016, pp. 779–788. [Online]. Available: <http://ieeexplore.ieee.org/document/7780460/>
- [35] M. Tan, R. Pang, and Q. V. Le, "EfficientDet: Scalable and Efficient Object Detection," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Seattle, WA, USA: IEEE, Jun. 2020, pp. 10 778–10 787. [Online]. Available: <https://ieeexplore.ieee.org/document/9156454/>
- [36] R. Pflugfelder, "An In-Depth Analysis of Visual Tracking with Siamese Neural Networks," Aug. 2018, arXiv:1707.00569 [cs]. [Online]. Available: <http://arxiv.org/abs/1707.00569>
- [37] J. F. Henriques, R. Caseiro, P. Martins, and J. Batista, "High-Speed Tracking with Kernelized Correlation Filters," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 3, pp. 583–596, Mar. 2015, conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.
- [38] M. Danelljan, G. Hager, F. S. Khan, and M. Felsberg, "Learning Spatially Regularized Correlation Filters for Visual Tracking," in *2015 IEEE International Conference on Computer*

- Vision (ICCV)*. Santiago, Chile: IEEE, Dec. 2015, pp. 4310–4318. [Online]. Available: <http://ieeexplore.ieee.org/document/7410847/>
- [39] D. S. Bolme, J. R. Beveridge, B. A. Draper, and Y. M. Lui, “Visual object tracking using adaptive correlation filters,” in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Jun. 2010, pp. 2544–2550, iSSN: 1063-6919.
- [40] M. Danelljan, G. Bhat, F. S. Khan, and M. Felsberg, “ATOM: Accurate Tracking by Overlap Maximization,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Long Beach, CA, USA: IEEE, Jun. 2019, pp. 4655–4664. [Online]. Available: <https://ieeexplore.ieee.org/document/8953466/>
- [41] G. Bhat, M. Danelljan, L. Van Gool, and R. Timofte, “Learning Discriminative Model Prediction for Tracking,” in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. Seoul, Korea (South): IEEE, Oct. 2019, pp. 6181–6190. [Online]. Available: <https://ieeexplore.ieee.org/document/9010649/>
- [42] S.-K. Weng, C.-M. Kuo, and S.-K. Tu, “Video object tracking using adaptive Kalman filter,” *Journal of Visual Communication and Image Representation*, vol. 17, no. 6, pp. 1190–1208, Dec. 2006. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1047320306000113>
- [43] X. Li, K. Wang, W. Wang, and Y. Li, “A multiple object tracking method using Kalman filter,” in *The 2010 IEEE International Conference on Information and Automation*, Jun. 2010, pp. 1862–1866.
- [44] M. Kristan, A. Leonardis, J. Matas, M. Felsberg, R. Pflugfelder, J.-K. Kamarainen, H. J. Chang, M. Danelljan, L. Zajc, A. Lukežič, O. Drbohlav, J. Bjorklund, Y. Zhang, Z. Zhang, S. Yan, W. Yang, D. Cai, C. Mayer, and G. Fernandez, “The Tenth Visual Object Tracking VOT2022 Challenge Results,” 2022.
- [45] A. Ali, A. Jalil, J. Niu, X. Zhao, S. Rathore, J. Ahmed, and M. Aksam Iftikhar, “Visual object tracking—classical and contemporary approaches,” *Frontiers of Computer Science*, vol. 10, no. 1, pp. 167–188, Feb. 2016. [Online]. Available: <http://link.springer.com/10.1007/s11704-015-4246-3>
- [46] M. Kristan, A. Leonardis, J. Matas, M. Felsberg, R. Pflugfelder, L. C. Zajc, T. Vojir, G. Hager, A. Lukežic, A. Eldesokey, and G. Fernandez, “The Visual Object Tracking VOT2017 Challenge Results.”
- [47] S. Javed, M. Danelljan, F. S. Khan, M. H. Khan, M. Felsberg, and J. Matas, “Visual Object Tracking With Discriminative Filters and Siamese Networks: A Survey and Outlook,” *IEEE Transactions*

on *Pattern Analysis and Machine Intelligence*, vol. 45, no. 5, pp. 6552–6574, May 2023, conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

- [48] D. Held, S. Thrun, and S. Savarese, “Learning to Track at 100 FPS with Deep Regression Networks,” *arXiv:1604.01802 [cs]*, Aug. 2016, arXiv: 1604.01802. [Online]. Available: <http://arxiv.org/abs/1604.01802>
- [49] E. Real, J. Shlens, S. Mazzocchi, X. Pan, and V. Vanhoucke, “YouTube-BoundingBoxes: A Large High-Precision Human-Annotated Data Set for Object Detection in Video,” in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Honolulu, HI: IEEE, Jul. 2017, pp. 7464–7473. [Online]. Available: <http://ieeexplore.ieee.org/document/8100272/>
- [50] Z. Zhu, Q. Wang, B. Li, W. Wu, J. Yan, and W. Hu, “Distractor-aware Siamese Networks for Visual Object Tracking,” *arXiv:1808.06048 [cs]*, Aug. 2018, arXiv: 1808.06048. [Online]. Available: <http://arxiv.org/abs/1808.06048>
- [51] Z. Zhang, H. Peng, J. Fu, B. Li, and W. Hu, “Ocean: Object-aware Anchor-free Tracking,” *arXiv:2006.10721 [cs]*, Jul. 2020, arXiv: 2006.10721. [Online]. Available: <http://arxiv.org/abs/2006.10721>
- [52] Z. Yang, Y. Wei, and Y. Yang, “Associating Objects with Transformers for Video Object Segmentation,” in *Advances in Neural Information Processing Systems*, vol. 34. Curran Associates, Inc., 2021, pp. 2491–2502. [Online]. Available: <https://proceedings.neurips.cc/paper/2021/hash/147702db07145348245dc5a2f2fe5683-Abstract.html>
- [53] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Kaiser, and I. Polosukhin, “Attention is All you Need,” in *Advances in Neural Information Processing Systems*, vol. 30. Curran Associates, Inc., 2017. [Online]. Available: <https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html>
- [54] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale,” Jun. 2021, arXiv:2010.11929 [cs]. [Online]. Available: <http://arxiv.org/abs/2010.11929>
- [55] M. Raghu, T. Unterthiner, S. Kornblith, C. Zhang, and A. Dosovitskiy, “Do Vision Transformers See Like Convolutional Neural Networks?” Mar. 2022, arXiv:2108.08810 [cs, stat]. [Online]. Available: <http://arxiv.org/abs/2108.08810>

- [56] S. Mittal, “A Survey on optimized implementation of deep learning models on the NVIDIA Jetson platform,” *Journal of Systems Architecture*, vol. 97, pp. 428–442, Aug. 2019. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1383762118306404>
- [57] S. Han, J. Pool, J. Tran, and W. Dally, “Learning both Weights and Connections for Efficient Neural Network,” in *Advances in Neural Information Processing Systems*, vol. 28. Curran Associates, Inc., 2015. [Online]. Available: <https://proceedings.neurips.cc/paper/2015/hash/ae0eb3eed39d2bcef4622b2499a05fe6-Abstract.html>
- [58] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, “Pruning Filters for Efficient ConvNets,” Mar. 2017, arXiv:1608.08710 [cs]. [Online]. Available: <http://arxiv.org/abs/1608.08710>
- [59] J.-H. Luo, J. Wu, and W. Lin, “ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression,” 2017, pp. 5058–5066. [Online]. Available: https://openaccess.thecvf.com/content_iccv_2017/html/Luo_ThiNet_A_Filter_ICCV_2017_paper.html
- [60] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen, “Compressing Neural Networks with the Hashing Trick,” in *Proceedings of the 32nd International Conference on Machine Learning*. PMLR, Jun. 2015, pp. 2285–2294, iSSN: 1938-7228. [Online]. Available: <https://proceedings.mlr.press/v37/chenc15.html>
- [61] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” Apr. 2015, arXiv:1409.1556 [cs]. [Online]. Available: <http://arxiv.org/abs/1409.1556>
- [62] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks,” 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. 525–542.
- [63] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1,” Mar. 2016, arXiv:1602.02830 [cs]. [Online]. Available: <http://arxiv.org/abs/1602.02830>
- [64] R. Caruana, A. Niculescu-Mizil, G. Crew, and A. Ksikes, “Ensemble selection from libraries of models,” in *Proceedings of the twenty-first international conference on Machine learning*, ser. ICML '04. New York, NY, USA: Association for Computing Machinery, Jul. 2004, p. 18. [Online]. Available: <https://doi.org/10.1145/1015330.1015432>
- [65] G. Hinton, O. Vinyals, and J. Dean, “Distilling the Knowledge in a Neural Network,” Mar. 2015, arXiv:1503.02531 [cs, stat]. [Online]. Available: <http://arxiv.org/abs/1503.02531>

- [66] A. Chaman and I. Dokmanic, “Truly shift-invariant convolutional neural networks,” in *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Nashville, TN, USA: IEEE, Jun. 2021, pp. 3772–3782. [Online]. Available: <https://ieeexplore.ieee.org/document/9577703/>
- [67] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, “Microsoft COCO: Common Objects in Context,” in *Computer Vision – ECCV 2014*, ser. Lecture Notes in Computer Science, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, pp. 740–755.
- [68] A. Lukežič, L. Zajc, T. Vojříř, J. Matas, and M. Kristan, “Now you see me: evaluating performance in long-term visual tracking,” Apr. 2018, arXiv:1804.07056 [cs]. [Online]. Available: <http://arxiv.org/abs/1804.07056>
- [69] Y. Wu, J. Lim, and M.-H. Yang, “Online Object Tracking: A Benchmark,” in *2013 IEEE Conference on Computer Vision and Pattern Recognition*. Portland, OR, USA: IEEE, Jun. 2013, pp. 2411–2418. [Online]. Available: <http://ieeexplore.ieee.org/document/6619156/>
- [70] M. Mueller, N. Smith, and B. Ghanem, “A Benchmark and Simulator for UAV Tracking,” 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. 445–461.
- [71] H. Fan, L. Lin, F. Yang, P. Chu, G. Deng, S. Yu, H. Bai, Y. Xu, C. Liao, and H. Ling, “LaSOT: A High-Quality Benchmark for Large-Scale Single Object Tracking,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Long Beach, CA, USA: IEEE, Jun. 2019, pp. 5369–5378. [Online]. Available: <https://ieeexplore.ieee.org/document/8954084/>
- [72] M. Müller, A. Bibi, S. Giancola, S. Alsubaihi, and B. Ghanem, “TrackingNet: A Large-Scale Dataset and Benchmark for Object Tracking in the Wild,” in *Computer Vision – ECCV 2018*, V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Eds. Cham: Springer International Publishing, 2018, vol. 11205, pp. 310–327, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-030-01246-5_19
- [73] M. Kristan, J. Matas, A. Leonardis, T. Vojir, R. Pflugfelder, G. Fernandez, G. Nebehay, F. Porikli, and L. Cehovin, “A Novel Performance Evaluation Methodology for Single-Target Trackers,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 11, pp. 2137–2155, Nov. 2016, arXiv:1503.01313 [cs]. [Online]. Available: <http://arxiv.org/abs/1503.01313>