WHITEPAPER

Optimizing Deep-Learning Inference for Embedded Devices

SILEXICA

IRIDA LABS

WHITEPAPER

Optimizing Deep-Learning Inference for Embedded Devices

Executive summary

Deep artificial neural networks (ANNs) have emerged as universal feature extractors in various tasks as they approach (and in many cases surpass) human-level performance. They have become fundamental building blocks of almost every modern artificially intelligent (AI) application, from online shop recommendations to self-driving cars.

As these networks are typically very computationally demanding, they are often deployed in large datacenters as a cloud back-end for edge devices (e.g. smartphones) that are unable to process input data in a reasonable time.

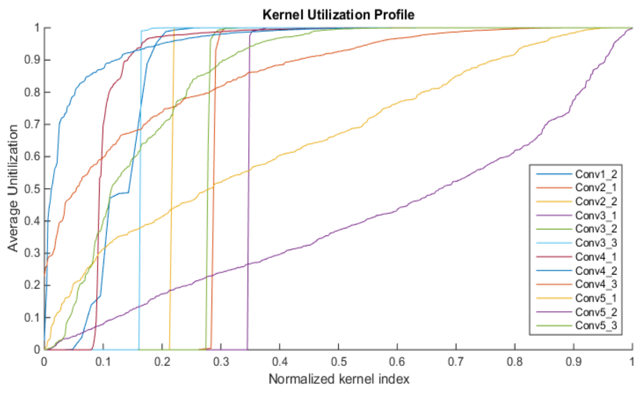
In embedded (real-time) applications where cloud-based processing is not an option, these networks must operate under strict timing constraints with limited computational resources. Here, the focus is on low latency and power efficiency and not on high (average) throughputs. For example, the stream of a front-mounted camera of a self-driving car must be processed in real-time with a low latency for other depending sub-systems (e.g. route planning) to be able to respond rapidly to the environment (e.g. to avoid collisions).

In comparison to a desktop application development, embedded devices typically require in-depth knowledge of different custom hardware architectures and programming interfaces. The tool-support for these devices is usually very limited. This problem is especially evident for modern embedded platforms that feature a heterogeneous set of processors like CPUs, GPUs, and DSPs.

In the case of ANNs, the major deep learning frameworks are mainly focused on a limited number of embedded CPUs or GPUs. Thus, the capabilities of modern SoCs for heterogeneous processing are often under-utilized, while optimization options are usually limited to leveraging fast matrix-to-matrix multiplications and algebraic-level simplifications.

The process of ANN design is mainly driven by recognition or detection accuracy while the characteristics of the target platform are only considered after a final ANN architecture has been selected. The selected ANN architecture therefore might not match the characteristics of the underlying hardware platform, potentially leading to performance issues and in the worst-case, time-expensive redesigns of the original architecture. Here, significant performance improvements can be achieved through network-level optimizations, e.g. by modifying the shape and size of typical ANN layers.

In this paper, it is demonstrated how a state-of-the-art single-shot object detection (SSD) ANN [1] can be optimized for embedded execution on the Nvidia Jetson 2 and the Nvidia Drive PX2 platforms.



SSD outputs contain both bounding boxes and different objects in an input image in a single execution. The overall architecture is composed of a back-end network responsible for extracting high-level features and a front-end network for processing features and pre-defined bounding box proposals. By exchanging the back-end network, a tradeoff between performance and accuracy can be achieved. In this paper, the VGG[2], PVA-Net [3], and SqueezeNet[4] back-end networks are evaluated in terms of computational complexity and accuracy. Additionally, it is demonstrated how the SSD architecture can be optimized without affecting its accuracy.

Table of Contents

1. Introduction

This whitepaper highlights how different challenges related to the deployment (inference) of artificial neural networks (ANNs) on embedded devices can be addressed. For this, the state-of-the-art object detection single-shot detector (SSD) is optimized for execution on the Nvidia Jetson X2 and Drive PX2 platforms. In the following, deep neural networks and the challenges related to training and inference are introduced.

1.1 Artificial Neural Networks (ANNs)

The nodes (neurons) in an ANN are typically arranged in different layers. For vision applications, the first ANN layers are usually convolutional layers (see Figure 1). Here the input signal (typically an RGB image) is convolved with a set of convolutional kernels resulting in a set of (so-called) feature maps. The weights of the convolutional kernels are determined during the training of the ANN. After the convolution, a trainable bias term is added. The output of each layer is usually further processed with operations that are not trainable and do not further expand the parameter space but are considered important structural elements of the overall model architecture. Typical examples of such operations are non-linear activation functions (e.g. sigmoid, ReLU, etc.), local response normalization functions, pooling and subsample operations, and element-wise algebraic operations with feature-maps (i.e. additions, subtractions, Hadamard products etc.).

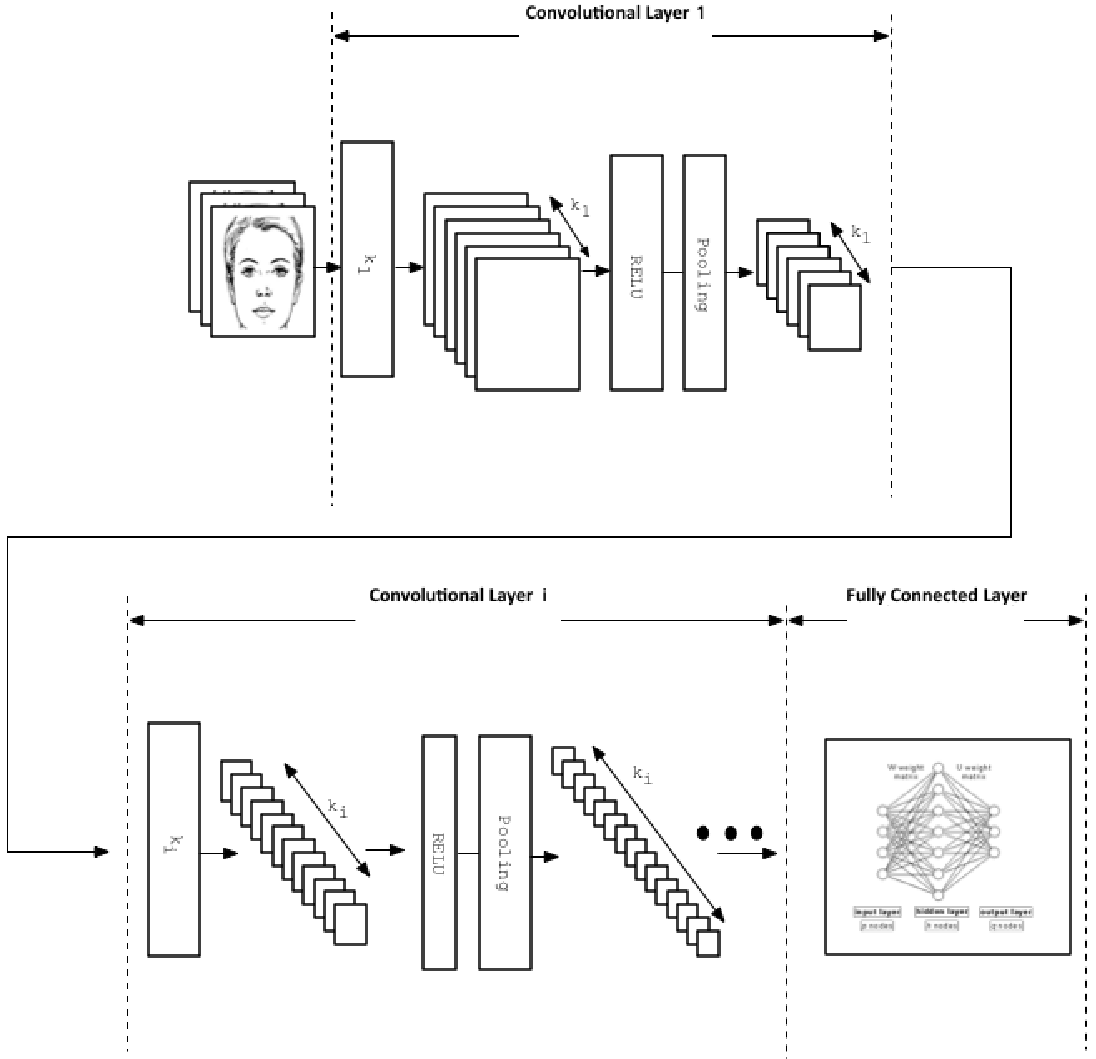


Figure 1: Typical structure of a convolutional ANN

A common practice is to utilize several fully-connected layers after the convolutional layers. The output of such a network is a vector of numbers that represent the estimation of the model based on the input signal. For an image classification scenario, these numbers represent the probability that the processed input image belongs to a specific class (e.g. the face of a specific person). For more complex applications such as object detection, a network can produce multiple outputs by performing inference for both classification and regression tasks (i.e. multi-scale recognition and localization).

In general, an ANN comprised of more than three layers is characterized as a "deep" network. For most use cases, the inference accuracy increases as the model gets deeper.

1.2 Training Challenges

During training, the selection of the network architecture, optimization algorithm, cost function, and the data acquisition/preprocessing need to be addressed. As ANNs appear in many forms and variants, architectural hyper-parameters are the easiest way a developer can modify and improve the accuracy for a given problem. Due to a lack of theoretical foundations and tools, a user typically must manually explore the ANN design space and select an optimal architecture (network depth, layer types, layer layout, layer sizes, etc.) using empirical criteria.

In most cases, users are restricted to work with existing architectures trained on large datasets, as the number of available data for a given task is not sufficient to train the network from scratch. To achieve this, the user uses a pre-trained model which is then incorporated (fully or partially) in the target network design. The model is then fine-tuned on the target training data to solve the optimization problem for the target task (transfer learning). While a rule of thumb is not available for the number of training data required to fine-tune a network, it has been observed that fine-tuning can be performed with significantly less data compared to training from scratch.

However, if the user wants to create a network architecture from scratch, transfer learning cannot be applied. This poses significant limitations as training typically requires enormous amounts of data, which in most cases is unavailable. Also, by training on the new task with insufficient amounts of data it is likely that the network will not be able to generalize well on new unseen data. In this regard, users are limited in the selection of ANN architectures.

In general, when transfer learning (or similar techniques) cannot be applied, training deep learning models is a time consuming and data intensive task which typically takes weeks (sometimes months) of time.

1.3 Inference Challenges

ANNs have the capacity to learn complex mappings between input data and target labels by iterating through large amounts of training data. Their learning capacity is related to the number of parameters and the depth (number of layers) of the network. Depending on the task, bigger/deeper networks typically perform better. This in turn leads to high computational demands and a requirement for high-performance computing systems for both training as well as deployment (inference).

Deep learning software frameworks have been developed to enable end-users to easily construct and train network models. These frameworks are designed to take advantage of the available computing infrastructure. While they are capable to generate code for inference, the computing libraries have been mainly developed for powerful modern desktop/datacenter systems.

Due to their superior performance for perception tasks (which typically happens on the edge), deep learning applications are used more frequently in embedded applications. Consequently, the embedded platforms in these scenarios carry the same computational load that was previously held by desktop/datacenter systems. In addition, they have to cope with additional constraints like low-latency execution and power consumption. Special computing libraries for embedded devices are missing or at best immature due to the large variability of architectures and diversity of programming tool-chains. Existing libraries are optimized for training and not for inference. This is an important issue when dealing with inference of a single datum (e.g. image), as the hardware is likely underutilized.

Some vendors like Nvidia provide tools to optimize ANN code for inference, however, for most available platforms this is not the case. Hence, the challenge of porting deep learning applications for edge applications results from both limited resources as well as a lack of optimized libraries or software tools.

As time to market plays a critical role in an increasingly competitive environment, the porting of ANNs can play a big role in the viability of a solution both from a technical as well from an economical perspective. Therefore, the need for automated tooling that streamlines the process of porting ANNs to embedded devices and allows early feedback during the design time of ANNs is required.

2. Materials and Methods

In this section, the single-shot object detector (SSD) ANN, the target platforms and the deep learning framework used in this paper are presented. Additionally, ANN-specific optimizations are explained in detail.

2.1 Single-Shot Object Detection (SSD) ANNs

Object detection is one of the most challenging and computationally demanding tasks in computer vision. In the past, a sliding window of variable size was used to evaluate a configurable number of image patches individually. A classifier (e.g. a support vector machine (SVM)) trained on these image patches was then used to assign categories to each patch. This approach however is limited as a large number of patches has to be evaluated for accurate object detection. Additionally, the features used to classify these patches are typically shallow (e.g. histograms of gradients, etc.) and thus not sufficient to efficiently describe the variability of the appearance of objects.

ANNs for object detection outperform these classic methods due to their ability to extract higher-level representations of data. These high-level representations are available in the form of feature maps (filter responses in the convolutional layers). Initially, a separate region proposal mechanism which allows to evaluate a relatively small number of patches was used. This approach however is computationally complex, especially for embedded and low-power devices. To cope with this complexity, single shot detection (SSD) ANNs were proposed which only require a single pass of the image. They detect objects for a predefined (fixed) number of windows of variable sizes.

In SSD ANNs the activation maps are directly evaluated by both a classifier and a regressor at one[5] or more [2] scales. In this paper, we use a multi-box SSD ANN[2] which evaluates feature maps at different scales and provides high accuracy with a reasonable performance overhead. The selected SSD consists of a back-end that is mainly used for classification and a front-end that is mainly used for region proposals and post-processing (non-maxima suppression, filtering, etc.).

The back-end network is responsible for creating feature maps that efficiently describe different objects and heavily influences the overall performance since both the accuracy and the complexity are mostly dictated by this part of the model. It can be freely exchanged, allowing for a trade-off between accuracy, computational complexity, and performance efficiency. To this end, in this paper, three different back-end networks (see Table 1) are evaluated in terms of performance and computational complexity.

	VGG-SSD	PVA-SSD	SqueezeNet-SSD
Computational Complexity (MACs)	31.4 G	1.3 G	1.2 G
Number of Parameters	26.3 M	5.7 M	5.5 M
Accuracy / mAP	77.2	68.7	64.2

Table 1: Computational complexity and accuracy for three SSD back-end networks.

For the selection of the appropriate back-end network, the target platform capabilities as well as the required accuracy (measured in mAP) must be considered. For example, VGG-SSD requires approximately 24 times more multiply-accumulate operations (MACs) than PVA-SSD to achieve a 12 % higher accuracy (in mAP).

The high computational complexity required for inference however make VGG-SSD unsuitable for embedded devices. Thus, other alternatives such as PVA-SSD and SqueezeNet-SSD should be considered. These two networks have roughly the same number of parameters, but PVA-SSD surpasses the accuracy of SqueezeNet-SSD by 4% in terms of mAP.

However, this accuracy improvement also leads to an increased computational complexity that cannot directly be attributed to the number of MACs. More specifically, PVA-SSD has a more complex computational graph which leads to increased execution times, while SqueezeNet-SSD follows a more straightforward approach leading to a faster implementation.

Object detection networks are typically trained on large, publicly available datasets like MS-COCO[1], KITTI[i], or Pascal-VOC[2]. For this paper, the Pascal-VOC dataset has been used. Each network was trained to detect 20 different object classes.

2.2 Embedded Platforms

As target platforms for the embedded optimization of the presented SSD ANNs, the commercially available Nvidia Jetson 2 and Drive PX2 AutoChauffeur platforms were selected.

The Nvidia Jetson 2 development board features the low-power Nvidia Tegra X2 system-on-chip (SOC). The Tegra X2 features an ARM-based CPU cluster and a powerful GPU (Pascal architecture) with 256 CUDA cores operating at up to 1465 MHz. It can achieve a throughput of up to 0.75 TFLOPS in FP32 (1.5 TFLOPS in FP16). The CPU consists of a Denver processor (ARM-based) with two-cores operating at up to 2.0 GHz and an ARM Cortex-A57 with four cores running at up to 2.0 GHz.

The SOC contains 8 GB LPDDR4 RAM that is shared between the CPU and the GPU. The max. power consumption is 15 W. The standard operating system for the Jetson 2 is a Nvidia variant of Ubuntu 16.04 (Nvidia Jetpack (Version 3.2)) with Cuda 9.0 and CuDNN 7.0 support. Due to the CUDA/CuDNN support, ANNs can typically be ported easily to the platform. The Tegra X2 is, for example, used in embedded industrial or robotic applications.
The Nvidia Drive PX2 AutoChauffeur platform, consists of two separate but identical subsystems. Each subsystem features a Tegra X2 SOC (see above) and a dedicated discrete GPU (dGPU, Pascal architecture). The dedicated GPU features 1152 CUDA cores running at up to 1290 MHz. In total, the DrivePX2 can achieve a throughput of up to 8 TFLOPS in FP32 (20 TFLOPS in FP16). The two subsystems are connected over an internal router and can in theory be configured to work on a single task. Additionally, an Infineon Aurix is used to configure/control the entire system.

The max. power consumption is 250W. As the name suggests, the Drive PX2 is targeted at the automotive market and can be found in Tesla vehicles. The standard operating system for each subsystem of the Drive PX2 is Nvidia Vibrante Linux which supports Cuda 8.0 and CuDNN 6.0. Similarly, to the Jetson 2, this allows the effective execution of typical ANNs.

2.3 Network Optimization

One way to improve the performance of ANNs on embedded devices without losing accuracy is the removal (pruning) of redundant operations/layers/information in the network. Most reduction approaches achieve this in an off-line, post-training approach. For this, the main approaches are kernel pruning and weight quantization.

In coefficient/kernel pruning, coefficients/kernels that do not contribute to the network response in a significant way (for most inputs) are removed completely from the network. This can, for example, be achieved in a data-driven approach where the output of neurons is evaluated for a large dataset [6]. Based on unlabeled data, the structure of a model can be estimated which allows to balance between model complexity and fidelity[7].

Structured sparsity learning can be used to regularize the number of filters (and their shapes), the number of channels, and the depth of the network[8]. By first training a network to learn which connections are important, subsequently removing unimportant connections and retraining the network, connections can be removed permanently without affecting accuracy [9].

1. http://cocodataset.org
2. http://www.cvlibs.net/datasets/kitti/
3. http://host.robots.ox.ac.uk/pascal/VOC/

In most of these methods, the coefficients of an ANN are analyzed after the training and some of them are zeroed according to their magnitude, leading to sparse matrices exploitable by sparse arithmetic software. In others, the ANN is trained to contain as many insignificant coefficients as possible (so that connections can be removed without affecting accuracy).

For weight quantization, the floating-point coefficients of a network are replaced by fixed point representations to which the input data and output responses are mapped [10].

Pruning and weight quantization can be utilized separately or in combination and typically lead to improved inference speed and reduced power consumption.

2.3.1 Parsimonious Inference Pruning

In this paper, a novel pruning approach ("Parsimonious Inference Pruning") by Irida Labs is used [11] to optimize the inference performance of the networks under investigation. The approach supports both static (permanent) and dynamic (during inference) kernel pruning. In comparison to other pruning approaches, the resulting models can be trained in hours and run faster on embedded devices without sacrificing accuracy.

The kernel pruning is based on so-called Learning Kernel Activation Map (LKAM) modules that are placed around convolutional layers (see Figure 2). The modules learn which filters are important for specific inputs. After the training phase, filters that were determined to be redundant for the majority of inputs can be pruned permanently without the loss of accuracy (static pruning).

Additionally, filters can be deactivated on-the-fly depending on the input. For this, the LKAM modules have to be integrated permanently into the ANN. In both cases, fewer operations must be executed typically resulting in improved performance at similar accuracy.

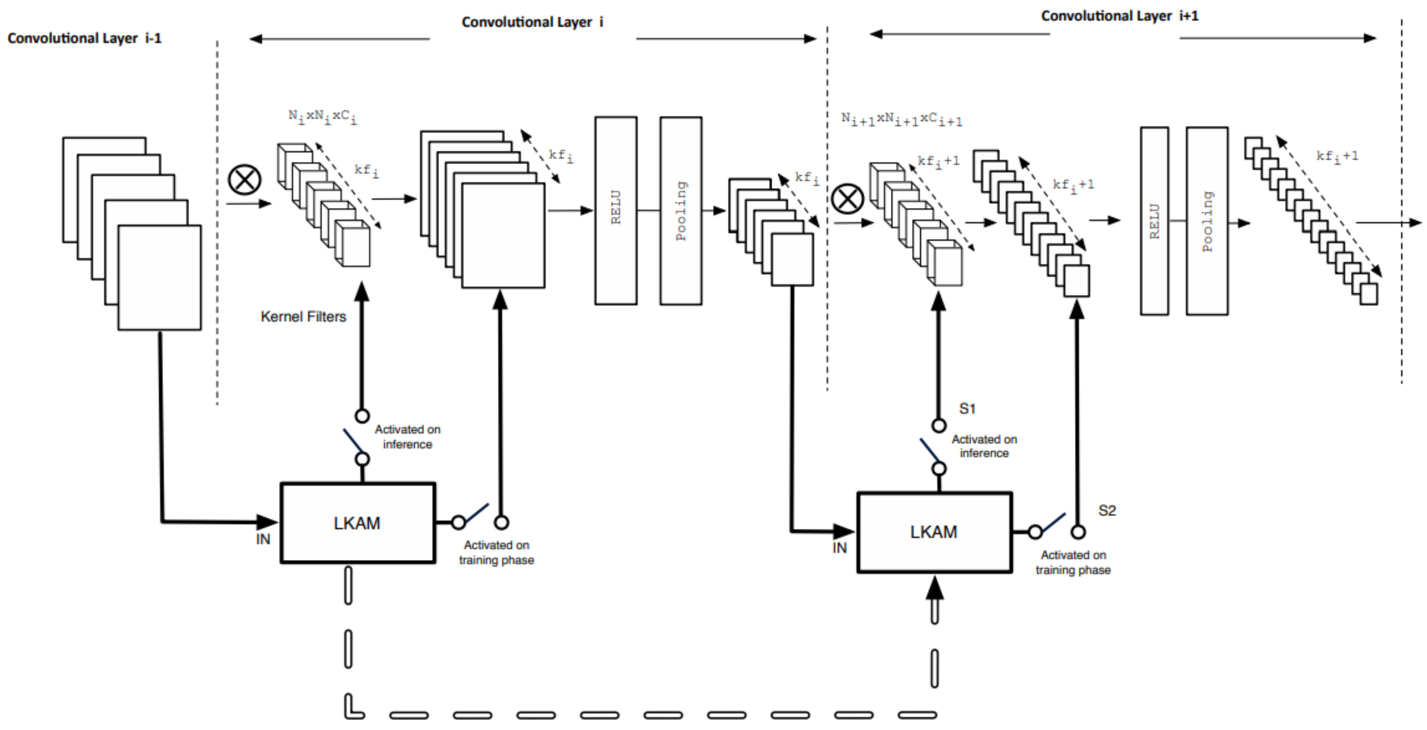


Figure 2: The Learning Kernel Activation Map (LKAM) module [11]

2.3.2 Dynamic Pruning

In the following, the pruning process is demonstrated for the VGG-SSD network. First, the LKAM modules are inserted around the convolutional layers. Next, the model is retrained to allow the LKAM modules to learn the redundant filter kernels. A detailed analysis of the utilization of the different convolutional kernels in VGG-SSD can be seen in Figure 3.

A significant percentage of convolutional kernels was turned off permanently during the evaluation (e.g. in layer "Conv3_3") and can therefore be removed safely from the network.

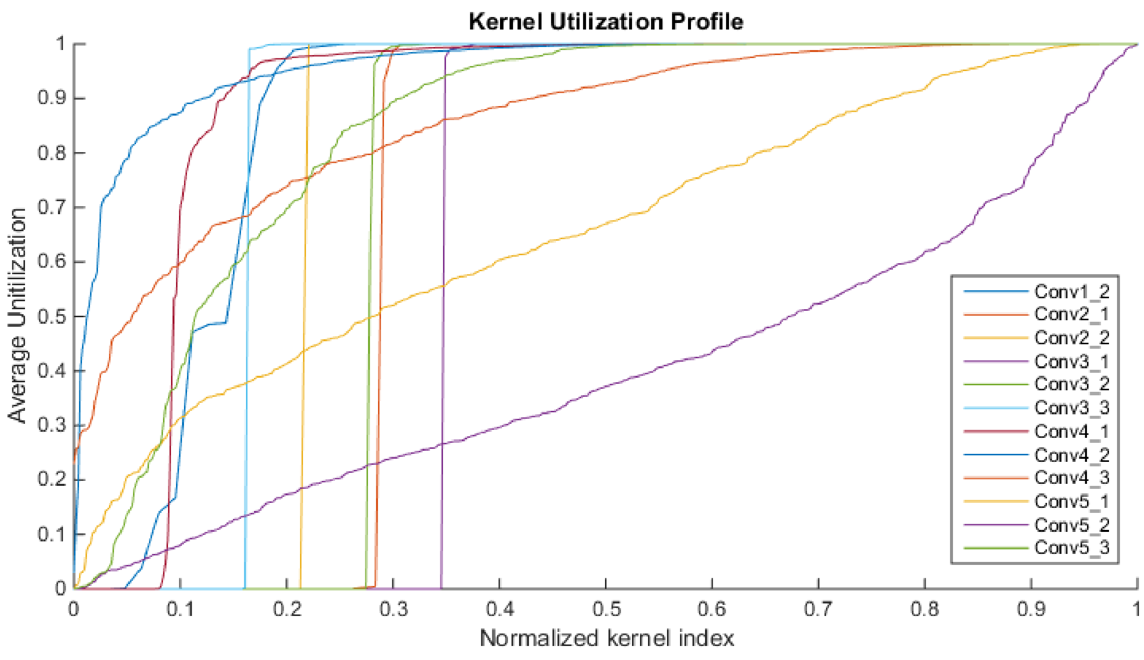


Figure 3: Kernel utilization profiles of the VGG-SSD Parsimonious Inference model

Overall Statistics	
Average reduction in OPs	31.5 % ± 2.6
Minimum reduction	26.4 %
Maximum reduction	49.6 %
mAP of emulated pruned model	73

Table 2: Overall Statistics

Through pruning, the computational load (measured in MACs) is reduced significantly and ranges from 50.1-73.6 % of the original load for the full network (see Table 2).

2.3.3 Static (Permanent) Pruning

As Figure 3 shows, a certain number of layers are not (or barely) contributing to the result. These layers can be removed permanently from the network without affecting its accuracy. Here, kernels that have an average utilization lower than 5 % are pruned permanently. After pruning, the remaining kernels are multiplied with their utilization coefficients and the model is fine-tuned using the same dataset to compensate for differences incurred by the changes. As a result, the computational complexity of the network is reduced. The results of the permanent pruning of kernels in the back-ends of the three SSD implementations can be seen in Table 3. While the focus here was on the optimization of the back-end layers, in some cases the front-end of SSD networks can also be optimized. For example, for the SqueezeNet-SSD the total number of MACs is reduced by 42% when both the front-end and the back-end are optimized.

	VGG-SSD			PVA-SSD			SqueezeNet-SSD		
Network Component	Original	Pruned	Diff. %	Original	Pruned	Diff%	Original	Pruned	Diff.%
Back-end MACs	30.1 G	23.4 G	22.3 %	976 M	706 M	27.4 %	535 M	397 M	25.8 %
mAP	77.7	77.4	-0.3%	68.9	68.4	-0.4%	64.2	64.8	+0.7 %

Table 1: Number of operations for back-end networks and model sizes for various SSD networks.

2.4 Caffe Deep Learning Framework

The open source Caffe[4] deep learning framework was explicitly designed for computer vision applications. It was the first deep learning framework that allowed users to design networks without writing any code. It is mainly implemented in C++ and features efficient GPU implementations for many common layer types.

It can easily be extended with new layers (which requires the creation of low-level C++/CUDA code however). Networks can be imported/exported as protocol buffer (protobuf) files.

Similar to other deep learning frameworks, Caffe lacks official support for embedded devices (Android is supported only unofficially). For embedded devices that are running minimal Linux operating systems (or non-Linux operating systems), Caffe is not supported as it has several library dependencies that prevent (or complicate) a port.

For this paper, the Caffe framework was selected as it is supports both optimized GPU (CUDA/cuDNN) code and optimized CPU code (using OpenBLAS). It is also the framework in which the original SSD ANN [2] was implemented.

3. Results

In this section, the results of the optimization of the different SSD variants on the Nvidia Jetson X2 and Drive PX2 are presented. Additionally, a real-time implementation of the different SSDs that processes live camera input and outputs the camera stream overlaid with detected objects is shown.

3.1 Caffe Timing Measurements

The inference times were measured using the internal Caffe timing utility. For the measurements, Wifi and Bluetooth were disabled, and the maximum performance mode was enabled on the Nvidia Jetson 2. Similarly, the maximum performance mode was enabled on the Nvidia Drive PX2.

The Parsimonious Inference (PI)-based network optimizations improve the performance on both CPU and GPU in all scenarios (see Table 4). On average, PI optimization improves the performance by 9.7 % on the Nvidia Drive PX2 and by 8.2 % on the Nvidia Jetson 2. For both platforms, the performance improvements are slightly higher for the CPU (13 % on the Drive PX2, 14.3 % on the Jetson 2) than for the (integrated) GPU (~10 % on both platforms).

On network level, the pruning is most effective for VGG-SSD, where the performance is up to 20.7 % higher for the pruned network. The smallest improvements can be seen for PVA-SSD on both platforms. This can be explained by the complexity of the model architecture which is composed of more than 400 nodes which stalls the execution.

In general, GPU performance is significantly (up to an order of magnitude) better than CPU performance for all networks. As could be expected, the Tegra X2 (CPU and integrated GPU (iGPU)) on the Drive PX2 performs similar to the Tegra X2 on the Jetson 2 board for all networks (with the Jetson 2 being slightly faster) while the discrete GPU of the Drive PX2 outperforms all other devices.

Nvidia Drive PX2									
	VGG-SSD			PVA-SSD			SqueezeNet-SSD		
	Original (in ms)	Pruned (in ms)	Change (in %)	Original (in ms)	Pruned (in ms)	Change (in %)	Original (in ms)	Pruned (in ms)	Change (in %)
CPU	1869.3	1532.6	- 18.0	286.3	278.9	-2.6	215.9	176.1	-18.4
iGPU	160.7	138.6	-13.8	67.4	62.4	-7.4	35.1	31.8	-9.4
dGPU	49.1	43.7	-11.0	56.8	56.1	-1.3	24.6	23.2	-5.5

Table 2: Inference times (in milliseconds) for the different networks on the Nvidia Drive PX2 (above) and the Nvidia Jetson 2 (below. Inference times are also included for the CPU and GPU. Lower numbers indicate higher performance.

Nvidia Jetson 2									
	VGG-SSD			PVA-SSD			SqueezeNet-SSD		
	Original (in ms)	Pruned (in ms)	Change (in %)	Original (in ms)	Pruned (in ms)	Change (in %)	Original (in ms)	Pruned (in ms)	Change (in %)
CPU	1806.8	1432.6	-20.7	281.7	261.1	-7.3	204.5	174.2	-14.8
GPU	157.1	136.1	-13.4	65.9	60.9	-7.5	34.8	31.4	-9.7

3.2 Demo

To illustrate and validate the findings, a live demo was prepared for the Embedded World trade fair held in Nuremburg in 2018. The image streams of four Logitech C920 HD cameras were processed by two Nvidia Jetson 2 boards (one camera each) and one Nvidia Drive PX2 (one camera per subsystem) with the goal to detect and classify objects in real-time. Each camera stream was processed using a different network.

Here, next to VGG-SSD, VGG-SSD-Pruned, and SqueezeNet-SSD, Tiny-Yolo-V2 [5] (which uses a different architecture than the SSD-based networks) was included for comparison.

The cameras were positioned on top of a 4K television screen (see Fig. 4) which displayed the live results of the object detection for all four cameras in a split-screen. For the execution, an SSD variant(3) and a Yolo-V2 variant(4,5) of Caffe were used.

3. https://github.com/weiliu89/caffe/tree/ssd
4. https://github.com/eric612/Caffe-YOLOv2-Windows (ported to Linux)
5. http://pjreddie.com/darknet/yolo (network, weights)

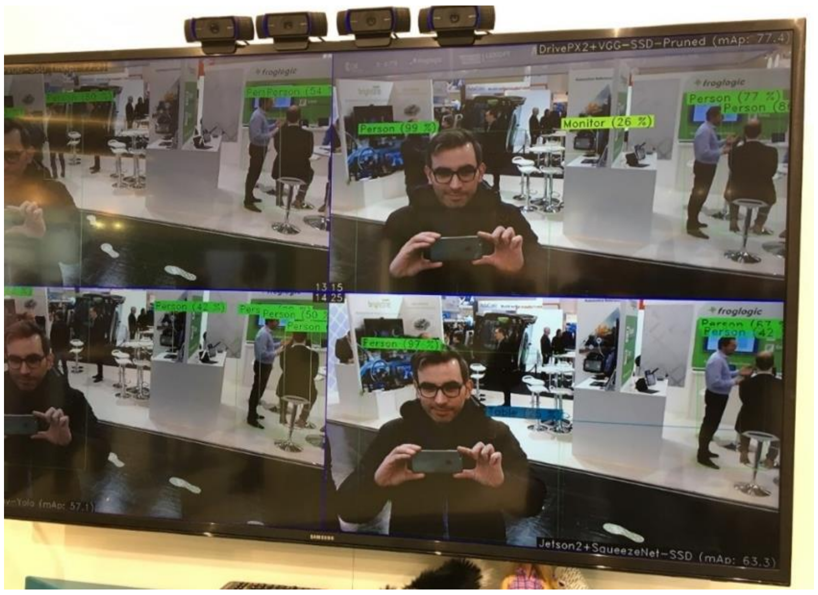

DrivePX2+VGG-SSD-Pruned (mAp: 77.4)
Person (99 %)
Monitor (26 %)
Person (77 %)
Person (97 %)
Jetson2+SqueezeNet-SSD (mAp: 63.3)
Yolo (mAp: 57.1)

Figure 4: Embedded World 2018 demonstration setup: The 4K television screen is split into four independent parts where each part displays the detected and classified objects of one camera stream.

To improve the demo performance and reduce latency, the processing was split into two threads. The threads exchange image frames through a free-running ring buffer. One thread reads frames from the camera and writes them to the ring buffer, while the second thread selects the most recent frame, processes it for objects, and displays the results on the screen. This approach significantly reduces the latency between the camera input and the output of images on the screen.

To reduce the impact of outside influences, Wi-Fi and Bluetooth were disabled. Also, the platform clocks were set to maximum using scripts provided by Nvidia. The performance for the different parts of the demo can be seen in Table 5. Next to the frame rate (frames per second (FPS)) also the accuracy (in mAP) and the size of the weights file (in MB) is given as a reference. Through pruning, the performance of VGG-SSD could be improved by 4 FPS (+28.6 %) without losing accuracy.

The impact of a smaller back-end network can be seen for the SqueezeNet-SSD variant running on a Jetson 2: Although the Jetson has significantly less computing power than the Drive PX2, the network can still be run in real-time (25 FPS) with a reasonable loss of accuracy (-16.7 %).

Name	**Platform**	**Accuracy (in mAP)**	**Frames per Second (FPS)**	**Weights File Size (in MB)**
VGG-SSD	Nvidia Drive PX2	77.2	14	105
VGG-SSD-pruned	Nvidia Drive PX2	77.4	18	100
Tiny-YoloV2	Nvidia Jetson 2	57.1	14	63
SqueezeNet-SSD	Nvidia Jetson 2	64.3	25	22

Table 5: Results for the Embedded World 2018 object detection demonstration.

4. Conclusions

In this paper, the challenges of porting deep learning (DL)-based applications to embedded platforms have been discussed. For a state-of-the-art, single-shot object detector (SSD) artificial neural network (ANN) it was shown, that optimizing deep learning applications for inference on embedded platforms requires optimizations that go beyond typical embedded performance optimization.

In this regard, it was shown that (neural) network-level optimizations can be used to improve the inference performance without affecting accuracy. For example, using Parsimonious Inference (PI) pruning, the inference performance of VGG-SSD in a live demo on the Nvidia Drive PX2 was improved by 28.6 % without affecting its accuracy.

The PI pruning can be used dynamically on a per input basis or statically (permanently removing connections from the network). By selecting a different back-end network (SqueezeNet), it was demonstrated that SSD ANNs can be run on embedded platforms (Nvidia Jetson 2) with a small power envelope (<15 W) in real-time (25 FPS) at a reasonable loss of accuracy (-16.7 %).

In general, more automated tooling is required that allows to systematically design and optimize ANNs for embedded platforms, at best, even before the time-consuming training phase.

About Silexica
Silexica provides software development solutions that enable technology companies to take intelligent products such as autonomous cars from concept to deployment. The SLX programming tools help developers implement software to run efficiently on embedded supercomputers by offering deep understanding of how software behaves on the system.

Silexica was founded in 2014 and has raised $28 million in funding. With headquarters in Germany and offices in the US and Japan, Silexica has partnered with global customers across many rapidly transforming industries including automotive, wireless and aerospace. For further information please visit: www.silexica.com

About Irida Labs
Irida Labs (IL) develops software for embedded computer vision for a variety of embedded computing platforms using state of the art Machine Learning and Deep Learning/Artificial Intelligence technology. The company specializes on algorithmic development, especially designed to respect strict specs regarding resource budget and develops software exploiting every available on-chip computational resource, in the context of what is called heterogeneous computing.

5. References

[1] W. Liu et al., “SSD: Single Shot MultiBox Detector,” CoRR, vol. abs/1512.0, 2015.

[2] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” CoRR, vol. abs/1409.1556,2014.

[3] K.-H. Kim et al., “PVANET: Deep but Lightweight Neural Networks for Real-time Object Detection,” CoRR, vol. abs/1608.08021, 2016.

[4] F. N. Iandola et al., “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size,” CoRR, vol. abs/1602.07360, 2016.

[5] J. Redmon and A. Farhadi, “YOLO9000: Better, Faster, Stronger,” CoRR, vol. abs/1612.0, 2016.

[6] H. Hu et al., “Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures,” CoRR, vol. abs/1607.03250, 2016.

[7] J. Feng and T. Darrell, “Learning the Structure of Deep Convolutional Networks,” in 2015 IEEE International Conference on Computer Vision (ICCV), 2015, pp. 2749–2757.

[8] W. Wen et al., “Learning Structured Sparsity in Deep Neural Networks,” CoRR, vol. abs/1608.0, 2016.

[9] S. Han et al., “Learning both Weights and Connections for Efficient Neural Networks,” CoRR, vol. abs/1506.02626, 2015.

[10] D. D. Lin et al., “Fixed Point Quantization of Deep Convolutional Networks,” CoRR, vol. abs/1511.0, 2015.

[11] I. Theodorakopoulos et al., “Parsimonious Inference on Convolutional Neural Networks: Learning and applying on-line kernel activation rules,” CoRR, vol. abs/1701.0, 2017.