Real-Time Object Detection On Low Power Embedded Platforms

George Jose Aashish Kumar Srinivas Kruthiventi
Sambuddha Saha Harikrishna Muralidhara
Harman International India Pvt. Ltd., Bangalore
{george.jose, aashish.kumar, srinivas.sai,
sambuddha.saha, harikrishna.muralidhara}@harman.com

Abstract

Low power real-time object detection is an interesting application in deep learning with applications in smart wearables, Advanced Driver Assistance Systems (ADAS), drone surveillance systems, etc. In this paper, we discuss the limitations with existing networks and enumerate the various factors to keep in mind while designing neural networks for a target hardware. Based on our experience of working with TI embedded platform, we provide a systematic approach for designing real time object detection networks on low power embedded platforms. First stage involves identifying the optimal layers for the hardware, by understanding it's computational and memory limitations. The next step is to use these layers to come up with a basic building block that has low computational complexity. The final stage involves using model compression techniques like sparsification/quantization to accelerate the inference process. Based on this design approach, we were able to come up with a low latency object detection model HX-LPNet that operates at 22 FPS on low power TDA2PX System on Chip(SoC) provided by Texas Instruments (TI).

1. Introduction

Object detection is one of the most popular research areas in deep learning today. The advent of GPUs with heavy computational power and memory capacity has boosted the use of Convolution Neural Networks(CNNs) for this task. Object detection is also one of the most challenging problems with applications in autonomous driving scenarios, surveillance systems, etc where it need to detect multiple objects in the image with varying sizes. A lot of recent research on CNNs has shown promising results for complex object detection tasks. But one major drawback is the need for high computational power and larger memory footprint associated with models like Faster RCNN [22], RetinaNet [13], etc. This limits its application in battery-operated low power systems like smart wearables, Advanced Driver Assistance Systems (ADAS), drone surveillance systems, etc. fuelling the need for developing low power detection architectures.

Tiny YOLO [19], SSDLite [6, 23] and SqueezeDet[28] are some of the existing architectures having low memory footprint and low computation power. Although these networks are widely used in mobile devices, we found it inefficient for our hardware platform. In this paper, we explain the various challenges faced by us to develop a neural network for object detection task on low power TDA2PX SoC provided by TI. We discuss in detail the drawbacks associated with the above networks with respect to the hardware platform used by us. Our work is inspired by JacintoNet11_v2 [16], a previous work done on this platform. We perform modifications to this network to come up with a low latency object detection model that operates at 22FPS. Based on these design experiences, we provide a systematic approach to design neural networks which can be extended to any other embedded platform. Some of these guidelines are similar to the ones suggested by the authors of ShuffleNetv2 [15].

The paper has been organized in the following manner. In section 2, we discuss the various research works in this area. Section 3 discusses the different hardware factors to be considered while designing a network on embedded platforms. Sections 4 and 5 gives a description of our hardware platform and design strategy used to develop a low complexity network on this hardware. Finally in section 6, we show the performance of our model in comparison with previous work done on this platform.

2. Related Work

In this section, we will be discussing various techniques currently adopted for reducing the memory footprint and accelerating the inference process in neural networks. First we will discuss the basic modules in various backbone networks like MobileNets [6, 23], ShuffleNets [30, 15], SqueezeNets [9, 28] which have been designed for performing inference in real-time. Then we will be discussing various network pruning techniques that are applied during training or post-training to reduce the number or precision of weights to accelerate the inference process. Finally, we discuss how neural architecture search (NAS) can be applied to get a compact model.

The following are the notations used in this paper. C_i is the number of input channels, C_o is the number of output channels, M and N are the width and height of the input feature map to the convolutional layer. For simplicity, we have taken a filter with kernel size 3×3.

2.1. Depthwise Separable Convolutions

Depthwise separable convolutions perform a low-rank approximation of $3 \times 3 \times N$ convolutional filter by decomposing it into two filters of sizes 3×3 (depthwise convolution) and a $1 \times 1 \times N$ (pointwise convolution). First, the 3×3 filter is convolved across each channel separately to learn spatial features and then the channel information is aggregated by sliding a 1×1 filter at each spatial location (see Fig 1c).

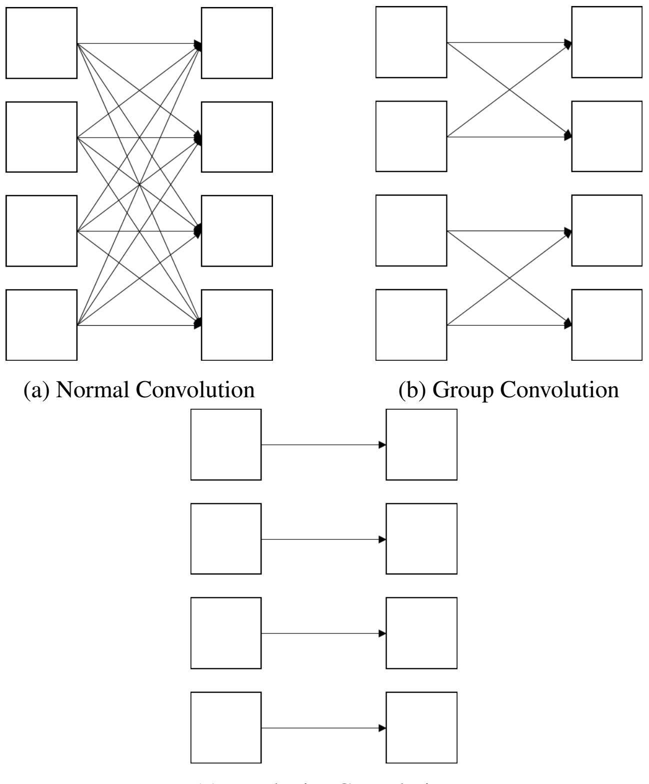
For each filter, the number of parameters required is reduced from $3 \times 3 \times C_i \times C_o$ to $(3 \times 3 + C_i)$. The number of computations for the entire layer is also reduced from $3 \times 3 \times C_i \times C_o \times M \times N$ to $(3 \times 3 \times C_i \times M \times N + C_i \times C_o \times M \times N)$. These depthwise separable convolutions are able to achieve 8 to 9 times reduction in computation at the cost of only a small reduction in accuracy. Some networks which use depthwise separable convolutions are MobileNets [6, 23], ShuffleNets [30, 15] and Xception [3].

2.2. Group Convolutions

The problem with the traditional convolutional filter is that it operates on the entire feature map depth, thereby increasing the computational complexity. Sometimes it might be redundant for each filter to operate across the entire depth of the feature map. Group convolutions solve this by splitting the input feature map across depth into different groups and applying different filters to each group (see Fig 1b).

For a layer with G groups, the input feature map will be split into groups with depth (C_i/G) channels. In order to produce an output feature map with depth C_o, each group will produce feature maps of depth C_o/G. Using group convolutions, the number of parameters is reduced from $(3 \times 3 \times C_i)$ to $(3 \times 3 \times C_i/G)$. The number of computations is also reduced to $(3 \times 3 \times C_i \times C_o \times M \times N)/G$. Thus there is a reduction in the total number of parameters and computations by a factor G to generate the same number of output channels. The first use of group convolution was in AlexNet [11] where it was used to distribute the model across GPUs.

Stacking group convolution layers has a drawback of limiting the information to a certain block of groups and not sharing across the entire channels. ShuffleNet solves this by simply shuffling the channels thereby spreading the information across different groups.


(a) Normal Convolution (b) Group Convolution

(c) Depthwise Convolution

Figure 1: Representation of input-output relationships for different types of convolutions. The left side represents input channels and the right side represents output channels. An edge between them represents that particular output channel is dependant on the corresponding input channel

2.3. Bottleneck Layers

Pointwise convolutions are used in bottleneck layers to reduce the computational complexity of traditional convolutional filter. Pointwise convolutions can be used to expand or squeeze the number of channels without changing the spatial resolution at a relatively low computational complexity. In bottleneck layers, first, the number of channels is reduced by applying 1×1 convolutions. The normal convolutional 3×3 filter is then applied on this feature map with reduced depth. The number of channels can then be increased by using a 1×1 pointwise convolution. The fire modules in SqueezeNet uses the bottleneck layer to reduce the number of computations. MobileNetv2 (see Fig 2b) uses a combination of depthwise separable convolution, and bottleneck layers to improve the performance of MobileNetv1.

2.4. Low Rank Filter Approximation

One approach towards low-rank filter approximation is to split the 3×3 convolutional kernel into two kernels of size 1×3 and 3×1. Thus, the number of parameters are reduced from $3 \times 3 \times C_i \times C_o$ to $(3 \times C_i \times C_o) \times 2$. This reduction will be more significant for larger kernels like 5×5 or 7×7 filters. Depthwise separable convolution can also be viewed as a low-rank approximation technique.

Another approach is to apply tensor decomposition as a post-training process on the learned weight matrix. The most commonly used techniques are Canonic Polyadic (CP)[12] and Tucker [10] decomposition in case of convolutional filters and truncated Singular Value Decomposition (SVD) in case of fully connected layers [4]. This will decompose the tensor weights into low-rank tensors.

2.5. Network Pruning

Neural networks are generally trained on large datasets with millions of parameters. Estimating the right amount of network complexity is a difficult task and often neural networks end up with a lot of redundant parameters. The rationale behind network pruning techniques is to remove this redundancy by reducing the capacity of the networks. One approach is to prune the network by removing individual neurons or entire convolutional filter based on some rank assigned to it[17]. Rank can be assigned based on l_1 or l_2 magnitude of the weights and the weights can be sparsified based on this. This can be done in an iterative manner where after sparsification, the model is retrained to compensate for the loss in accuracy [2]. Another approach is to reduce the precision of weights and activations [8]. The most common approach is to quantize the weights to 8 bits, but some works have shown that it can be reduced to even 1 bit [20] [7].

2.6. Neural Architecture Search

Until recently, state-of-the-art neural networks are hand-designed by human experts using heuristics combined with trial and error. However, there is an emerging trend of Neural Architecture Search (NAS) [31] which aims to discover optimal architecture for neural networks in an automated fashion. Several approaches based on reinforcement learning [31], evolutionary algorithms [21] and continuous optimization [14] are proposed to perform this search effectively. NASnets are shown to achieve the best trade-off between computational complexity and performance for tasks such as image classification, object detection. More recently, Cai et al.[1] have proposed to incorporate the computational latency of inference on the target platform into the search objective while discovering neural architecture.

3. Design Considerations For Hardware

In the following sections, we will discuss the various factors to be considered while designing neural networks for specific hardware.

3.1. Factors Affecting Computation Speed

In order to understand the performance limitations of different neural network architectures, its necessary to have an understanding of how a simple algorithm is executed on hardware. Following is a simplified model of the various stages involved in the execution pipeline :

1. Reading the inputs from memory
2. Performing the required math operations
3. Storing the outputs back to memory

Let T_{mem} be the time taken for accessing the memory and T_{op} be the time taken for executing the arithmetic operations. For a single-threaded execution without any parallelism, the total time taken for the entire operation is (T_{mem} + T_{op}). In practice, processors use pipe-lining mechanisms to improve hardware utilization. While one thread will be performing math operations, other threads would have already started fetching data for the next cycle. This ensures that resources do not stay idle. In this scenario, the total time taken to complete the process can be viewed as max(T_{mem}, T_{op}). Modern processors are equipped with multiple cores to incorporate more parallelism.

So the performance is limited depending on which of the above two operations takes the maximum duration. A process is math limited if the time taken for performing math operations are longer . A process becomes memory limited if the time taken for fetching and storing the memory becomes the bottleneck. When it comes to running deep neural networks on any platform, two major components that determine the performance are: a) Processing capability typically measured in the number of Floating Point Operations/seconds (FLOPs) b) Memory bandwidth and latency.

GPUs are designed to maximize the FLOPs by performing parallel calculations using multiple cores. Latest NVIDIA GTX 1080Ti has nearly 3500 CUDA cores operating at around 1.5GHz which is capable of delivering 11.3 TFLOPs. In comparison to this, our hardware TDA2PX SoC is equipped with two EVEs which can deliver around 28 GFLOPs at a clock frequency of 900MHz. The internal memory bandwidth of GTX1080Ti is around 27.3 Gbps, while for EVE processor in our SoC it is around 384Gbps.

3.2. Arithmetic Intensity of an Algorithm

An algorithm can be math limited or memory limited depending on its implementation in the hardware, memory requirements and processors bandwidth. T_{mem} of an algorithm is the amount of memory in bytes accessed divided by the memory bandwidth of the processor. T_{op} is the number of operations required for performing the algorithm divided by the processors math bandwidth. So an algorithm is a math limited if $T_{op} > T_{mem}$. This can be expanded as :

$$\left(\frac{\#ops}{BW_{math}}\right) > \left(\frac{\#bytes}{BW_{mem}}\right)$$

On rearranging the above expression, we get :

$$\left(\frac{\#ops}{\#bytes}\right) > \left(\frac{BW_{math}}{BW_{mem}}\right)$$

The LHS of the above equation is dependent only on the given algorithm and is known as the algorithms arithmetic intensity. The RHS is dependent on the processors memory and math bandwidth and sometimes referred to as ops: byte ratio [27]. Layers like convolutional layers and fully connected layers typically involve many calculations per input value and mostly belong to the category of math limited layers. Layers like pooling, activation functions, concatenation, addition, scale, bias perform only a few operations per bytes accessed making it memory limited. [18] provides an in-depth analysis of how the choice of parameters of the layer can make it memory-limited or time-limited.

3.3. Low Power Requirements

NVIDIA GTX 1080Ti GPUs operating at maximum capability requires a power of around 250W, while typical CPUs consume around 65W. For battery-operated devices like smart wearables and ADAS systems, this is unacceptable. Low power devices should typically consume power in the range of single digits to operating for a longer duration.

One major factor which affects energy consumption is the amount of off-chip DRAM (Dynamic RAM) memory access. Accessing DRAM memory consumes around twice the order of energy compared to on-chip SRAM(Static RAM) access. SRAM which is used for CPU cache has a higher memory bandwidth, but the cost limits the storage capacity of SRAM. DRAM which requires a lesser amount of transistors to store the same amount of data is cheaper but slower. Storing data in the cache is based on two main strategies:
Temporal locality: Data just accessed is kept in the cache assuming it is likely to be used again. This promotes the need for data reuse like in convolution, where loaded filter weights will operate on the entire spatial dimensions before being removed from SRAM.
Spatial locality: Neighbouring data is also loaded into cache, assuming it is likely to be used in the future. This promotes the need for storing data in an efficient manner. For example in the case of matrix multiplication, it will be more efficient to store the first matrix in row-major format and second matrix in column-major format.

Another way to reduce DRAM access is to reduce the number of filter weights and feature maps so that they can be stored entirely in the SRAM. Larger activation maps or filter weights will force the data to switch between SRAM and DRAM, thereby increasing energy consumption [15].

4. System Description

We use TDA2PX System-on-Chip (SoC) developed by Texas Instruments designed for efficient low power Advanced Driver Assistance Systems (ADAS) applications [25]. The TDA2PX SoC has a multiprocessor architecture comprising of two TMS320C66x digital signal processors (DSPs), VisionAccelerationPaC, Arm Cortex-A15 MPCore, and dual-Cortex-M4 processors. The vision processing functions are provided mainly by the two Embedded Vision Engines (EVEs). EVEs are equipped with a $32-bit$ RISC core for efficient program execution and a vector coprocessor (VCOP) for vision processing. The TDA2PX also has interfaces to integrate multiple peripherals like multi-camera interfaces, displays, CAN and GigB Ethernet AVB.

TI deep learning (TIDL) library [26] enables inference using deep neural networks on TDA2PX SoC. TIDL helps achieve efficient inference on the device by utilizing the hardware resources optimally and also provides support for sparse convolutions.

The development flow for performing deep learning-based applications can be categorized into three phases:
a) Model Training: The computational capacity of TDA2PX SoC is not sufficient to perform the training of deep learning models. Model is first trained offline on powerful GPUs using frameworks like Tensorflow or Caffe.
b) Model Translation: This tool takes the floating-point model trained using the above frameworks and converts to a fixed-point model (supports 4-bit to 12-bit) which is compatible by the TIDL library. This quantization speeds up the inference process. In addition, it provides the option to place individual layers on different processors (EVEs/DSPs) for further optimization. For example, convolutional layers are optimal on EVEs, while fully connected layers and detection layers are optimal on DSPs.
c) Model Inference: This final stage is responsible for running the quantized network model on the TDA2PX SoC using the application programming interfaces provided by the TIDL library.

5. Design Strategy For TDA2PX SoC

In this section, we will discuss a systematic approach we followed to find an optimal network on TDA2PX SoC. Our network HX-LPNet performs low latency object detection at $FPS > 20$ for an image of size 1024×512.

5.1. Identifying the optimal layers for hardware

Our design strategy was to first explore the commonly used blocks for lightweight object detection like MobileNets, SqueezeDets, and ShuffleNets on TDA2PX SoC. On experimentation we found these architectures to be slow on our platform. Table 1 shows the benchmarking of MobileNetv1, ShuffleNetv1 and JacintoNet11_v2 [16] on single EVE core running at 650MHz. SqueezeNet even with lower GMACs has a higher latency compared to JacintoNet11_v2 which is a modified version of Resnet10 without skip connections.

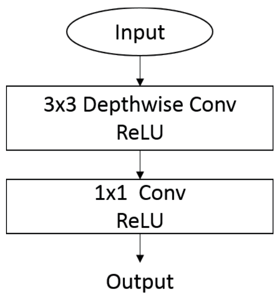
MobileNet based architectures use depth-wise separable convolutions as the basic building block. As discussed in section 2.1, a depthwise separable convolution is a combination of depthwise convolution and 1x1 pointwise convolution (see Fig 2a). Although the number of computations is less for this block, the major bottleneck is the memory access time for depthwise convolutions. The algorithmic intensity for this layer is low, since for each channel only a single filter is applied. Hence it behaves very similarly to an element-wise operation with memory access cost being the bottleneck.

SqueezeDets which are based on fire modules uses pointwise convolutions to reduce the channel size (squeeze layer). This is followed by an expansion module which is a mixture of 1x1 and 3x3 filters whose outputs are concatenated to produce a tensor with large channel size (see Fig 2c). Even though the number of computations is less compared to normal convolutions by squeezing the channels initially, the problem lies in the use of concat layer which can create an additional memory access cost, especially for large activation maps. Additionally, the SqueezeDets will have more layers with decreased complexity. For such an architecture, the latency can be high due to increased memory access cost. ShuffleNetv2 [15] also presents a similar observation, where they observe an increase in latency when the number of groups is increased.

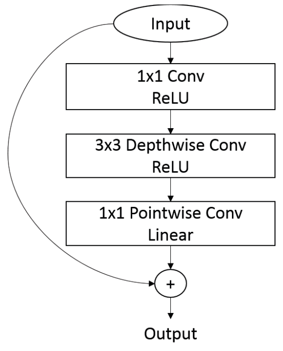
ShuffleNets uses a combination of pointwise convolutions, channel shuffle, depth-wise convolution along with a residual connection (see Fig 2d). Depthwise convolution, shuffle operation and residual connections are operations with low arithmetic intensity, thereby causing additional latency. Additionally, the shuffle layers are not supported by the TIDL library.

From the above observations, it can be inferred that layers like element-wise sum, concat, depthwise convolutions with low arithmetic intensity can hinder the performance due to high memory access cost. This further justifies the claim that along with the number of computations, one needs to compute the memory access cost also while designing for maximum performance in platforms constrained by memory bandwidth and compute power.

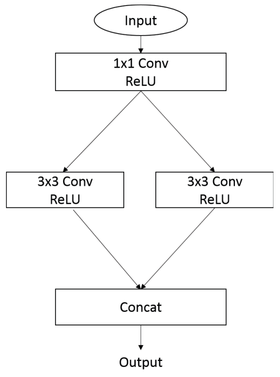
For our SoC, we found that the most optimal way to reduce computation was by the use of group convolutions. In our architecture, we mainly use groups of 2 and 4. The maximum group size used was 8 at a single place in the network where the depth was maximum. Since the channel shuffle layer is not supported by the TIDL library, we use a 1x1 pointwise convolution after stacked group convolution layers to use the aggregate channel information.



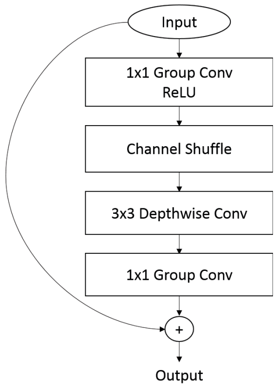
(a) Mobilenetv1



(b) Mobilenetv2



(c) Squeezenet



(d) Shufflenetv1

Figure 2: Basic building blocks of various commonly used lightweight object detection architectures

Network topology	Image Size	MMACs	Latency
Mobilenetv1	224x224	567.70	559.18ms
Squeezenetv1	227x227	390.80	237.60ms
Jacintonet11_v2 (D)	224x224	405.81	203.23ms
Jacintonet11_v2 (S)	224x224	107.54	103.23ms

Table 1: Benchmarking data by TI of various backbone architectures on a single EVE core at 635MHz for TDA2PX SoC. (D) represents dense model and (S) represents sparse model

5.2. Striking a balance between spatial dimensions and channel depth

For most of the networks, a major part of the computational complexity lies in the initial part of the network owing to its large spatial dimensions. Apart from increase in computational complexity, this will also introduce problem of having large intermediate feature maps. As discussed in section 3.3, large activation maps will force switching of data between DRAM and SRAM which will increase the energy consumption.

In the proposed backbone architecture, we try to maintain the computational complexity of layers nearly same throughout the network. In order to achieve this we need to find the right balance between spatial resolution and channel depth at a specific layer without compromising much on the accuracy. For initial layers, less number of channels were used ($C = 8, 32, 64$) to reduce the complexity. Once the spatial dimensions were reduced using max pooling layers, we use channels of 64 and 128. The maximum depth used in the network was 256 only at one layer, from where the detection heads are starting.

ID	Layer Type	Kernel Size	# O/p Channels	Stride	Groups
1	Conv,Relu	5	8	2	1
2	Conv,Relu	3	32	2	1
3	Maxpool	2	32	2	
4	Conv, Relu	3	32	1	2
5	Conv, Relu	3	64	1	4
6	Maxpool	2	64	2	
7	Conv,Relu	3	64	1	2
8	Conv,Relu	3	64	1	2
9	Conv,Relu	3	128	1	2
10	Conv,Relu	3	128	1	4
12	Conv,Relu	1	256	1	1
13	Conv,Relu	3	128	1	8

Table 2: Layer structure of our backbone feature extractor network

5.3. Use of sparse convolutions and quantization

This section discusses the techniques adopted to accelerate the inference process for the fixed network architecture. We mainly adopt two strategies for this:

Sparse convolutions: TIDL library provides support for sparse convolutions. In order to get sparse coefficients, a three stage training is adopted as explained in [16]. First is the normal training with l_2 regularization. Following this, we train a model using l_1 regularization which is known to introduce sparsity by bringing the weights closer to zero. In the final stage, sparsification is obtained by setting weights to zero less than a threshold which is determined dynamically based on the range of weights. From table 3, it can be observed that with 80% sparsity a 4x times improvement in speed can be obtained on TDA2Px SoC.

Sparsity (%)	# MegaCycles
80	8.95
75	12.91
50	23.35
30	31.62
20	35.98

Table 3: Comparison of % Sparsity vs # Mega Cycles required to complete $[128 \times 64 \times 64] \rightarrow 3 \times 3$ convolution $\rightarrow [128 \times 64 \times 128] \rightarrow BiasAdd + RELU \rightarrow [128 \times 64 \times 128]$ with $8 - bit$ layer parameters on a single EVE core at 635MHz. (Note: Data obtained from TIDL library datasheet)

Dynamic 8-bit quantization: Networks weights were trained with 32bit floating point precision. For inference, the TIDL model translation tool converts the model weights to 8-bit fixed representation based on the dynamic quantization method [16, 5]. The thresholds are determined separately for each layer. A similar strategy is adopted for model activations also. This reduction in precision helps increase the inference speed without much compromise in accuracy.

6. Results

6.1. Dataset Description

For experiments, we use the Berkeley DeepDrive Dataset, which is also known as BDD100K [29]. It consists of over 1000 hours of driving data videos collected from various locations in the United States under diverse weather conditions. Each video which is $40sec$ long was captured at $30fps$ with a resolution of $720p$. For our experiments, we only used images captured at daytime and under clear weather conditions. Further, the number of object categories was reduced from 10 to 3 (car, pedestrian, and truck).

6.2. Detection Model

For object detection, we used SSD based architecture (see Fig3) with our proposed backbone as a feature extraction module. Predictions were done at six different feature resolutions with feature map sizes varying from 64×32 to 4×2. Original 1280×720 images were resized to 1024×512 and fed to model for training. We compare our results with JDetNet [24], which is also an SSD based architecture with Jacintonet11_v2 as the feature extractor. As per benchmarking done by TI, dense Jacintonet11_v2 was able to operate faster than MobilenNet [6] and SqueezeNet [9]. After sparsification, JacintoNet11_v2 is around 2x times faster than SqueezeNet model (see Table1) with a latency of 107ms. Caffe-Jacinto framework[1] was used to train both models. This is a fork of NVIDIA Caffe with support for training sparse and quantized models. As described in section 5.3, both models were trained in 3 stages: a) initial training with l_2 regularization (b) training with l_1 regularization (c) training for sparsity. For the initial training, models were trained with a base learning rate of 1e-2 and weight decay of 0.0005. For l_1 and sparsity training, the base learning rate used was 1e-3. For all the experiments, Adam optimizer was used.

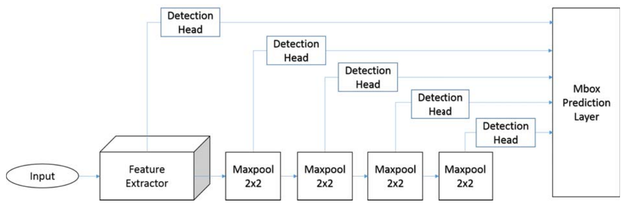


Figure 3: SSD Architecture

6.3. Model Evaluation

To evaluate the accuracy of various object detection models, we used mean Average Precision (mAP) metric. A model with higher mAP will perform better in detecting objects more accurately. Apart from this we also use latency and FPS to compare the performance of the model on hardware. An ideal model running on an embedded platform should be able to detect objects with high mAP, at low latency and high FPS.

Model	FPS	Latency	GMACs	mAP	%Sparse
JDetNet [24]	9.19	0.50s	4.49	63	34.90
HX-LPNet	22.47	0.20s	0.74	52	53.55

Table 4: Comparison of FPS, GMACs and mAP of JDetNet vs HX-LPNet while running on 2 EVEs and single C66x DSP core

For all our experiments, we use 2 EVE cores @ 635MHz and single DSP core @ 850MHz. All layers except the detection layer were placed in EVEs. Floating-point operations are required for the ArgMax layer which is faster in DSP compared to EVE. For evaluation, the images were fed to TDA2PX over Ethernet from a PC.

Table 4 shows the performance comparison of our model HX-LPNet with JDetNet. It can be observed that ours is a lightweight detection model with 6x fewer computations required compared to JDetNet. Dense JDetNet model has 3.15M parameters, while our dense model has only 0.42M parameters. Our sparse model was able to achieve 22.47 FPS on TDA2PX with a latency of only 0.2ms. This throughput and latency are much better than the JDetNet model. But these performance gains come with the cost of a reasonable reduction in overall mAP. On analysis of results, we found that our models have poor performance in detecting small objects. This is mainly due to the reduction in the number of channels in our model compared to JDetNet. But for ADAS applications we are mainly interested in detecting closer objects, which are mostly larger in size. Table 5, shows the mAP of our model at various stages of training and during inference with quantization. It can be observed that sparsification and dynamic quantization have little impact on the mAP. Compared to the dense model, the sparse model has half the parameters (53% sparsity) and FPS was observed to increase from 16.79 to 22.47. Even the reduction in precision from 16-bit while training to 8-bit for inferencing has a little impact on mAP. To conclude, sparsification and quantization are two good techniques that can be adopted to improve the speed and reduce the memory footprint of neural networks without much drop in accuracy. In terms of power consumption, the TDA2PX SoC has been designed to operate at single-digit power for ADAS applications.

Training Stage	mAP	FPS	Latency
Initial with l_2 reg	53.79	16.79	0.28s
Effect of l_1 reg	54.00	16.90	0.28s
Effect of sparsity	52.51	22.47	0.20s
Effect of quantization	52.00	22.47	0.20s

Table 5: Effect of l_1 regularization, sparsification, and quantization on model mAPs. Latency and FPS shown above are after quantizing the respective models inorder to run on TDA2PX SoC

7. Conclusion

In order to design a neural network on hardware with given performance requirements, it is necessary to understand the limits of the hardware first. Based on our experience, we provide one systematic approach: a) Finding the optimal layers for the hardware, by understanding its processor and memory limitations. b) Based on the above knowledge, find a basic building block that has low computational complexity that could be replicated to generate the network. Care should be taken to reduce the size of activations so as to reduce energy consumption, c) Use model compression techniques like sparsification/quantization to

[1]https://github.com/tidsp/caffe-jacinto

accelerate the inference process.

A drawback associated with this method is the amount of manual work involved in the first two stages which can lead to delays in development. Another good approach would be to use NASNets which aims at automating the process of finding the network design by taking into consideration hardware parameters like latency and energy consumption. More research in this direction can significantly reduce the timeline for developing efficient networks on embedded platforms.

References

[1] H. Cai, L. Zhu, and S. Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv:1812.00332*, 2018.

[2] G. Castellano, A. M. Fanelli, and M. Pelillo. An iterative pruning algorithm for feedforward neural networks. *IEEE transactions on Neural networks*, 8(3):519–531, 1997.

[3] F. Chollet. Xception: Deep learning with depthwise separable convolutions. In *IEEE conference on computer vision and pattern recognition*, 2017.

[4] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, 2014.

[5] P. Gysel. Ristretto: Hardware-oriented approximation of convolutional neural networks. *arXiv:1605.06402*, 2016.

[6] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861*, 2017.

[7] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks. In *Advances in neural information processing systems*, 2016.

[8] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[9] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv:1602.07360*, 2016.

[10] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv:1511.06530*, 2015.

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012.

[12] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv:1412.6553*, 2014.

[13] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. Focal loss for dense object detection. In *IEEE international conference on computer vision*, 2017.

[14] H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. *arXiv:1806.09055*, 2018.

[15] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *European Conference on Computer Vision*, 2018.

[16] M. Mathew, K. Desappan, P. Kumar Swami, and S. Nagori. Sparse, quantized, full frame cnn for low power embedded devices. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2017.

[17] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv:1611.06440*, 2016.

[18] NVIDIA. Deep learning sdk documentation. https://docs.nvidia.com/deeplearning/sdk/dl-performance-guide/index.html.

[19] J. Pedoeem and R. Huang. Yolo-lite: a real-time object detection algorithm optimized for non-gpu computers. *arXiv:1811.05588*, 2018.

[20] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, 2016.

[21] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. In *AAAI Conference on Artificial Intelligence*, 2019.

[22] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, 2015.

[23] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[24] Texas Instruments. Caffe jacinto models. https://github.com/tidsp/caffe-jacinto-models.

[25] Texas Instruments. TDA2Px evaluation module. http://www.ti.com/tool/TDA2PXEVM.

[26] Texas Instruments. TI Deep Learning Library Overview. https://training.ti.com/texas-instruments-deep-learning-tidl-overview.

[27] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009.

[28] B. Wu, F. Iandola, P. H. Jin, and K. Keutzer. Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2017.

[29] F. Yu, W. Xian, Y. Chen, F. Liu, M. Liao, V. Madhavan, and T. Darrell. Bdd100k: A diverse driving video database with scalable annotation tooling. *arXiv:1805.04687*, 2018.

[30] X. Zhang, X. Zhou, M. Lin, and J. Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[31] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv:1611.01578*, 2016.