Sparse, Quantized, Full Frame CNN for Low Power Embedded Devices

Manu Mathew, Kumar Desappan, Pramod Kumar Swami, Soyeb Nagori
Texas Instruments
Bangalore, India
www.ti.com
mathew.manu@ti.com, kumar.desappan@ti.com, pramods@ti.com, soyeb@ti.com

Abstract

This paper presents methods to reduce the complexity of convolutional neural networks (CNN). These include: (1) A method to quickly and easily sparsify a given network. (2) Fine tune the sparse network to obtain the lost accuracy back (3) Quantize the network to be able to implement it using 8-bit fixed point multiplications efficiently. (4) We then show how an inference engine can be designed to take advantage of the sparsity. These techniques were applied to full frame semantic segmentation and the degradation due to the sparsity and quantization is found to be negligible. We show by analysis that the complexity reduction achieved is significant. Results of implementation on Texas Instruments TDA2x SoC [17] are presented. We have modified Caffe CNN framework to do the sparse, quantized training described in this paper. The source code for the training is made available at https://github.com/tidsp/caffe-jacinto

1. Introduction

The computational complexity of full frame CNN applications is extremely high. For example it takes 402 Giga multiply accumulations per second (GMACS) to do the inference of AlexNet at 1280x720@30FPS. This kind of complexity is out of reach for typical low power embedded devices such as Digital Signal Processors (DSP), which are typically constrained to power consumption in single digit Watts. For CNN inference to be feasible on typical DSPs, the compute requirement has to come down below 50 GMACS. Our work presented here provides several tools to achieve complexity reduction. We show with an example that complexity can be reduced to a level where it becomes suitable for embedded implementation. We believe that the same methods will generalize to other networks and applications. In this section, we introduce the motivation for our work. The core concepts and results achieved are explained in the subsequent sections.

CNNs have become extremely popular due to their capacity to learn and generalize. Krizhevsky et al. [1] showed significantly higher classification accuracy in ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012 compared to classical approaches. This was the beginning of the transition in interest from traditional machine learning approaches such as Support Vector Machines and Decision Tree Classifiers to CNNs. Even then it was not obvious whether CNNs will generalize to other similar or different tasks. However, due to the works of Ross Girshick [2], Evan Shelhamer et al. [3], and several others, CNNs have been shown to do equally well on other tasks. Examples include object detection, semantic segmentation, and optical flow estimation. These are increasingly being applied for image recognition applications at datacenters, cars, mobile phones, medical equipment and in several other areas.

Many of these techniques require CNNs to be operated on a huge number of pixels per second. These systems are also sensitive to power consumption as well. Some of these applications require processing of high resolution video streams of 1 Mega pixel or higher – an example is Advanced Driver Assistance Systems (ADAS) object detection systems, where the range (distance) at which an object can be detected is directly related to the resolution. Many ADAS systems that process the image near to the camera (vs that processes the image at a central location) require the power consumption to be in single digits. On datacenters, it is necessary to process a large number of images in a second, which drives up the MACS required. Datacenters are also extremely conscious about the efficiency, which is usually measured in terms of Giga Operations per Watt (GOPS/W) of Terra Operations per Watt (TOPS/W).

As shown in Table 1, for popular CNN network configurations, the complexity of processing a video stream at 1 Mega pixel resolution can be quite high. For example an analysis in [9] showed that, in practice, an 8-core TMS320C6678 DSP [10] from Texas Instruments is capable of doing nearly 80 single precision GFLOPS while consuming 10 Watts for doing Level-3 BLAS operations. Since BLAS operations are popularly used to implement CNN convolutions, this may be taken as a representative number to analyze CNN operations as well.

However, the GFLOPS achieved in this case is nowhere close to the requirement of popular networks. Hence further research is needed to come up with network configurations that re suitable for low power embedded platforms.

CNN Network	GMACS
AlexNet [1]	389
CaffeNet	643
ResNet10 [7]	481
ResNet18 [6]	1005
ResNet50	1962
VGG16 [8]	8456

Table 1. MAC requirement for popular CNN models to process a video stream at 1280x720@30FPS (Excluding the fully connected layers).

2. Related work

There have been various approaches to reduce the complexity of CNNs. In this section we discuss some of the important ones that are relevant to this work.

2.1. Separable convolutions

One of the earlier approaches involved replacing the non-separable convolutions with separable convolutions [11]. This method claims to achieve about 4.5x speedup, with less than 1% drop in accuracy. One drawback of this method is that it changes the network structure by splitting a layer into separate horizontal and vertical filters. Also results on large datasets such as ILSVRC ImageNet need to be analyzed clearly.

2.2. Binary and Ternary operations

XNOR-Net developed by Mohammad Rastegari et al. [4] claims a 58x speedup in convolution operations by replacing multiplications with bit-wise XNOR operations. But the quality degradation reported for this method may not be acceptable for many applications. Binary Weight Networks developed by the same authors can help to speedup CNNs by using additions or subtractions instead of multiplications. Ternary Weight Networks by Fengfu Li et al. [5] improved upon this idea by using three levels for weights, and this gave better results. However these approaches can speed up CNNs only on platforms where additions and subtractions are significantly faster than multiplications and hence, are suitable for FPGA or ASIC implementations. On general purpose DSPs where multiplications usually take similar cycles as additions and subtractions, it may not result in significant speedup.

2.3. Sparse convolution

Yet another approach involves sparse convolution algorithms [12]. We found this approach to be quite interesting because, it doesn't change the overall network structure. A network is sparsified by training in iterations with high weight decay, and whenever the absolute value of a weight falls below predefined threshold, it is thresholded to zero. It also presented a method to recover the lost accuracy by fixing the zeroed out coefficients as zeros and fine tuning the non-zero coefficients. The sparsity induced is structurally constrained to get speedup in GPUs that use matrix multiplications to implement convolutions. In devices that do not use matrix multiplications for implementing convolution, structural constraint may not be beneficial. Also the sparsification approach used in the paper is extremely slow and time consuming – making it difficult to use for large datasets.

2.4. Quantization

Studies have shown that quantization of floating point coefficients to dynamic 8-bit fixed point is sufficient to retain the accuracy in Image classification problems as shown in Caffe Ristretto [13] and in Tensorflow [14]. 'Dynamic' in this context refers to the fact that the quantization multiplication factor, range etc. may change from layer to layer, between input, weights and outputs within a layer, and in the case of Tensorflow, from frame to frame as well. Caffe Ristretto used quantization with quantization multiplication factors restricted to power of 2, and mapped the range between -128 to +127 irrespective of whether the tensor being quantized is negative or not. The actual range consumed may be less than this because the quantization multiplication factor is restricted to power of 2, and the values may be unsigned. It also uses a fixed pre-computed range, decided at training time. The reported accuracy loss was 2.35% for GoogleNet. Tensorflow claimed that close to 4x speedup, 4x reduction in model size and considerable power savings can be obtained by quantizing from floating point to 8-bit with 1% drop in accuracy [15][16]. But it is also more complex than Ristretto. Tensorflow takes the original range of the tensor and maps it into an unsigned range of 0 to 255, thereby providing the best possible linear quantization. However, this is more complex and involves additional operations when there is a need to convert from one range to another, which is often the case between layers. It also uses on-the-fly computation of range for each input image. This is quite accurate, but suffers from the drawback of having to read the every tensor once completely before it is quantized. The un-quantized version of the output of a convolution layer is stored in 32-bit precision and then read it back to compute the range before applying the quantization to output. These additional memory accesses are not friendly for low power embedded devices.

3. The proposed approach

Our method involves the following steps. (1) a quick sparsification method (2) Fine tuning without loss of sparsity (3) Low complexity dynamic 8-bit quantization (4) Sparse convolution method to speedup inference.

3.1. Quick sparsification

As observed in the reference that induces structural sparsity [12], L1 regularized training is good at inducing sparsity of coefficients. So one of the first steps is to do an L1 regularized training with a relatively high weight decay. This will help us to distinguish more important coefficients from the less important ones and will make the job of thresholding easier.

Our quick sparsification is an empirical method that involves looking at the weights of each convolution layer separately and doing the following:

- *Let T_s be the sparsity target for a layer. T_s is the ratio of number of zero weights to the total number of weights in that layer. The sparsity target can be different for each layer.*
- *Find the maximum of the absolute value of weights in the layer. Let it be W_{am}.*
- *Set maximum threshold value, T_m to be a fraction α of W_{am}. We used an α value of 0.2.*

$$T_m = \alpha * W_{am} \tag{1}$$

- *Start with a small sparsity threshold t (which is much smaller than T_m), and check the amount of sparsity achieved (i.e. fraction of zero weights in the layer). If the sparsity is less than T_s, then increase the sparsity threshold t by a small value. Repeat this until the threshold t becomes greater than or equal to T_m or the sparsity target T_s is reached.*

```
W: weight tensor of the layer being thresholded
t=0;
s = Sparsity(W, t);
while s < Ts and t < Tm;
    t =t+ β
    s = Sparsity(W,t);
Where;
Sparsity() is a function that computes the sparsity of the given tensor at the threshold specified; it counts the number of coefficients having absolute value less than t.
β is a small increment value, example: 1e-7.
```

(2)

In most cases, the sparsity target is reached while obeying the constraint of maximum threshold value. In some cases, usually observed in the initial layers, the sparsity may be less than what is desired. Using a low value for maximum threshold is important for retaining accuracy.

As this thresholding step doesn't involve any training, it is quite fast and can be completed in few seconds.

3.2. Fine tuning without loss of sparsity

Theresholded model is seen to have lower accuracy than the original (not thresholded) model. However, most of the quality lost can be recovered by fine tuning, without loss of sparsity. It is done as in [12] by not allowing the already zero coefficients to change during the fine tuning. During the beginning of fine tuning stage, a map representing all non-zero coefficients is created. Only the non-zero coefficients are updated in the update step after the backpropagation, during the entire fine tuning process.

3.3. Low complexity dynamic 8-bit quantization

The quantization approach used in this work uses a middle ground between complexity and accuracy. Similar to Ristretto's approach, the minimum and maximum ranges of various tensors (weights, inputs and outputs) are computed for all the layers during training time, from a subset of the training data. Exponential moving averages are used during the training to compute these ranges. Once these ranges are computed, we determine if each tensor is signed or unsigned. Signed tensors are to be quantized between -128 and +127; while unsigned tensors are to be quantized between 0 and 255. Flags are inserted in our quantized CNN model to indicate whether these tensors are signed or unsigned. This is an improvement over Ristreto and will result in better accuracy of inference. However, we restrict the quantization multiplication factors to be a power of 2. This will allow us to do range conversion from one range to another by using only shifts. These shifts can be easily incorporated into the convolution itself in the case of the convolution layer. This saves additional multiplications required for range mapping (used in the case of Tensorflow). Also, since signed ranges are used for signed quantities, there are no additional offset adjustments required.

Since the ranges are computed from a subset of the training data and also since a moving average is used, the intention is not to find the worst case range, but to find a representative range. However, some values in a given tensor can exceed the computed range and have to be clipped appropriately during quantization. Given the ranges R_{min} and R_{max} of a tensor, the integer length of an unsigned tensor is computed as follows:

$$I_l = \log_2(|R_{\max}|) \tag{3}$$

The integer length of a signed tensor is computed as follows:

$$I_l = \log_2(\max(|R_{min}|, |R_{max}|)) + 1 \quad (4)$$

The fractional length is then computed as:

$$F_l = 8 - I_l \quad (5)$$

The quantization multiplication factor for tensor is calculated as:

$$M_q = 2^{F_l} \quad (6)$$

The tensor is quantized by multiplying with M_q and then clipped to the appropriate unsigned or signed range. For an unsigned tensor:

$$W_q = clip(round(W * M_q), 0, 255) \quad (7)$$

And for a signed tensor, it is done as follows:

$$W_q = clip(round(W * M_q), -128, +127) \quad (8)$$

Where *clip()* functions restricts the values to the given range and *round()* function converts the floating point value to integer value by a rounding operation.

3.4. Sparse convolution for inference

A pseudo-C code for a regular 3x3 convolution is shown as follows, where X is the input tensor, Y is the output tensor and W is the weight tensor. Wx, Hx are the width and height of X respectively.

```
for(int i=0; i<Wx;i++) {                                    (9)
 for(int j=0; j<Hx; j++) {
  for(int m=-1;m<=1;m++) {
   for(int n=-1;n<=1;n++){
      Y[i][j] +=  X[i+m]*[j+n] * W[m][n]
   }
  }
 }
}
```

However the order of these loops can be re-arranged so that the inner loops operate as multiplication of an entire block or a plane of data by a single weight value. The advantage of this re-arrangement is that the multiplication of the entire block can be skipped if the weight corresponding to that block is 0.

```
for(int m=-1;m<=1;m++) {                                    (10)
 for(int n=-1;n<=1;n++){
  wt=W[m][n]
  if(wt != 0) {
   for(int i=0; i<Wx;i++) {
    for(int j=0; j<Hx; j++) {
     Y[i][j] += X[i+m][j+n] * wt
    }
   }
  }
 }
}
```

We call this kind of convolution by the name Block Multiply Accumulation (BMA) as it operates on one block of data at a time. The approach above described convolution operation of a single 3x3 filter.

However it is possible to use BMA and complete the entire convolution layer. An analysis stage is first performed to first collect the block pointers corresponding to the blocks that have non-zero weights. These non-zero weights and the corresponding input and output pointers are collected in separate lists. Then these lists are passed to the BMA kernel which does the entire convolution layer. Here X, Y and W are assumed to be three dimensional, as these operations involve all the input and output channels. BMAList() function takes a block pointer from xList, a weight from wList and multiplies and accumulates the entire block into the block pointed to by the pointer in yList.

```
z=0;                                                        (11)
for(p=0;p<P;p++) { //output channels
 for(l=0;l<L;l++) { //input channels
  for(int m=-1;m<=1;m++) {
   for(int n=-1;n<=1;n++){
    if(|W[l][m][n]| > eps) {
     xList[z] = &X[l][m][n];
     wList[z] = &W[l][m][n];
     yList[z] =&Y[p][0][0];
     z=z+1;
    }
   }
  }
 }
}
BMAList(xList, wList, yList)
```

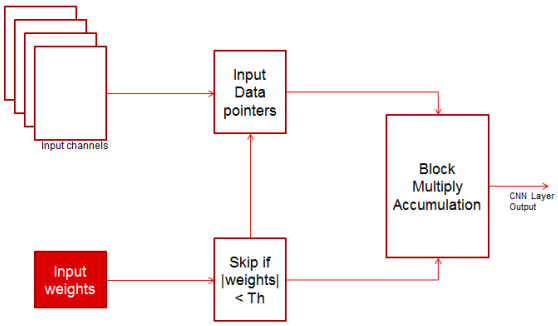

Figure 1. Convolution layer using Block Multiply Accumulation (BMA)

Note that the block size can be chosen appropriately to suite the local memory or data cache available in the system. In that case the whole image will have to be split into several ROIs and each ROI will have to be processed independently.

3.5. Analysis

An analysis was done to understand how much gain can be achieved in convolution layers by using BMA sparse inference. This analysis considers the data band-width available for loading data into the CPU along with the compute capability. The results of this analysis are given in Figure 2. We analysed two scenarios for quantization of convolution outputs after sparsification: (1) quantizing to 16-bit and (2) quantizing to 8-bit. It can be seen that quantizing to 8-bit provides much higher speedups at high sparsity when compared to 16-bit. So for our further experiments, we chose 8-bit quantization. As can be seen from the graph, at 85% sparsity a speedup (gain factor) of 4.5x can be expected from convolution layers. Although most of the compute in CNN is in convolution layers, the overall speedup of the entire network is expected to be slightly lower than this.

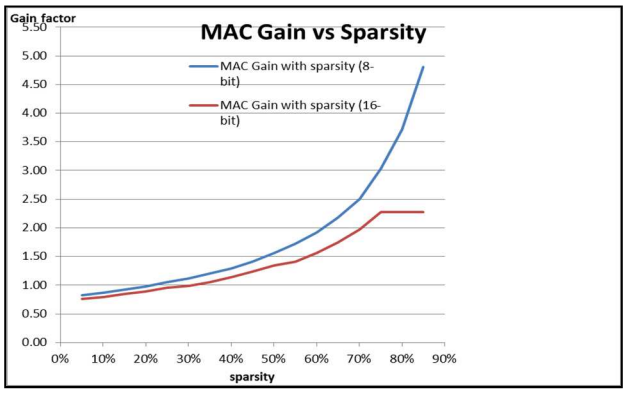

Figure 2. Analysis of potential speedup using sparse convolution.

4. Experiments and results

The sparsification and quantization method described in this paper (except the BMA inference which was implemented in the embedded SoC) was added to our custom Caffe fork. The model was trained on this modified version of Caffe. The source code and examples are made available at *https://github.com/tidsp/caffe-jacinto*

4.1. Network configuration

The base classification network was inspired by ResNet10 [7]. The ResNet10 network architecture was modified with the following changes.

- Residual connections are removed since it doesn't help much at small depths such as 10 as observed in the original ResNet paper [6]. Added groups of 4 to every alternate layer to reduce complexity. Grouped convolutions also help in data bandwidth reduction. It was first introduced in AlexNet.
- Max pooling is used instead of strides. We call the resultant network as JacintoNet11. More details are given in Table 2
- This network is used to train on the 1000 class ImageNet dataset.

Layer No	Layer type	Kernel size	Output channels	Stride	Group, Dilation
1	Conv,Relu	5	32	2	1,1
2	Conv,Relu	3	32		4,1
3	Maxpool	2		2	
4	Conv,Relu	3	64		1,1
5	Conv,Relu	3	64		4,1
6	Maxpool	2		2	
7	Conv,Relu	3	128		1,1
8	Conv,Relu	3	128		4,1
9	Maxpool	2		2	
10	Conv,Relu	3	256		1,1
11	Conv,Relu	3	256		4,1
12	Maxpool	2		2	
13	Conv,Relu	3	512		1,1
14	Conv,Relu	3	512		4,1
15	Avgpool	7			
16	FC	1			

Table 2. Layer structure of JacintoNet11 classification network

Then we train for Cityscapes [18] semantic segmentation using the pre-trained weights. For deriving the network for segmentation, the following changes are made:

- Removed the pooling in 5th block and changed the subsequent 3x3 convolutions to dilated ones as in the Dilation method of segmentation [19]. Added context convolution blocks and deconvolution layers as used in the same method.
- We call the resultant network as JSegNet21, since it has 21 convolutional and deconvolution layers. Most

of the complexity of this network is concentrated in layers 13 and 14 because the Maxpool stride before these layers is removed. Further details are given in Table 3.

Layer No	Layer type	Input Layer No	Output Channels	Kernel Size, Stride, Group, Dilation
1	Conv,Relu		32	5,2,1,1
2	Conv,Relu		32	3,1,4,1
3	Maxpool			2,2,-,-
4	Conv,Relu		64	3,1,1,1
5	Conv,Relu		64	3,1,4,1
6	Maxpool			2,2,-,-
7	Conv,Relu		128	3,1,1,1
8	Conv,Relu		128	3,1,4,1
9	Maxpool			2,2,-,-
10	Conv,Relu		256	3,1,1,1
11	Conv,Relu		256	3,1,4,1
12	Maxpool			**1,1,-,-**
13	**Conv,Relu**		512	3,1,1,**2**
14	**Conv,Relu**		512	3,1,4,**2**
15	Conv,Relu	14	64	3,1,2,4
16	Deconv		64	4,**2**,64,-
17	Conv,Relu	8	64	3,1,2,1
18	Eltwise	16,17		
19	Conv,Relu		64	3,1,1,1
20	Conv,Relu		64	3,1,1,4
21	Conv,Relu		64	3,1,1,4
22	Conv,Relu		64	3,1,1,4
23	Conv,Relu		8	3,1,1,1
24	Deconv		8	4,**2**,8,-
25	Deconv		8	4,**2**,8,-
26	Deconv		8	4,**2**,8,-
27	Argmax			

Table 3. Layer structure of JSegNet21 segmentation network

4.2. Training procedure

A smaller subset of 5-classes was selected for semantic segmentation. 32 classes of cityscapes were converted into 5-classes - so the trained model would learn to segment 5-classes (background, road, person, road signs, and vehicle). This 5-class training is different from the typical 19-class training done for cityscapes and reported on the benchmark website. This change was done to address a minimal scenario required for on-road classification for ADAS applications. The overall training procedure is as follows:

- ImageNet classification training for JacintoNet11
- L2 regularized Cityscapes training of JSegNet21 using pre-trained weights of JacintoNet11
- L1 regularized training. L1 regularization is a powerful technique in inducing several small coefficients. As shown in the final results, this is a powerful technique to induce sparsity (many of these small coefficients will become zero during quantization).
- Quick sparsification using thresholding. The first and last convolutions layers in the network were given a lower sparsity target as they have very few convolution coefficients. For example when sparsity target is 85%, the first and last convolution layers were a given a target of only 55%.
- Fine tuning to recover the quality lost during thresholding. In this stage and in the following training stages, coefficients that are already zero are not updated during back propagation update.
- Fine tuning with 8-bit quantization.

4.3. Sparsity measurements

Table 4 shows the sparsity achieved after various stages involved in the training. Quantization was done for each stage to collect the sparsity statistics (Quantization in the intermediate stages was used only for the purpose of statistics collection – the subsequent stage in training uses un-quantized weights).

Layer No	Layer type	Sparsity obtained with quantization		
		Initial L2 regularized training	L1 regularized training	Induced 80%
1	Conv,Relu	11.71	21.75	41.54
2	Conv,Relu	9.46	19.57	79.82
3	Maxpool			
4	Conv,Relu	9.90	38.02	80.00
5	Conv,Relu	3.73	21.93	79.97
6	Maxpool			
7	Conv,Relu	7.79	52.97	80.05
8	Conv,Relu	5.82	42.68	79.99
9	Maxpool			
10	Conv,Relu	8.14	66.31	82.71
11	Conv,Relu	6.51	61.59	80.12
12	Maxpool			
13	Conv,Relu	11.60	84.11	88.10
14	Conv,Relu	5.84	91.27	94.00
15	Conv,Relu	2.97	72.33	82.65
16	Deconv			
17	Conv,Relu	2.65	52.38	80.26
18	Eltwise			
19	Conv,Relu	2.39	47.04	79.99
20	Conv,Relu	2.18	46.39	79.99
21	Conv,Relu	2.29	47.42	80.01
22	Conv,Relu	2.28	54.19	80.00
23	Conv,Relu	38.85	53.93	47.94
24	Deconv			
25	Deconv			
26	Deconv			
27	Argmax			

Table 4. Sparsity achieved after various stages in training – with quantization

4.4. Accuracy

Change in accuracy due to sparsification and quantization is reported in Table 5. It is seen that the pixel

accuracy loss is almost negligible and the change in Mean IOU loss is reasonable. Sample images are shown indicating example segmentations in Figure 3.

Configuration	80% sparsity induction	
	Pixel Accuracy (%)	Mean IOU (%)
Initial L2 regularized training	96.20	83.23
L1 regularized fine tuning	**96.32**	**83.94**
Sparse, fine tuned	96.10	82.86
Sparse, Quantized (8-bit dynamic fixed point)	**95.90**	**82.15**
Overall impact due to sparsification and quantization	**-0.42**	**-1.79**

Table 5. Impact of sparsification and quantization, for semantic segmentation on the Cityscapes dataset.

4.5. Implementation on device

These sparse convolutions were implemented on the TDA2x SoC [17] from Texas instruments. It is a low power SoC that operates in single digit Watts of power. It has 4 Embedded Vision Engines (EVEs), which are co-processors suited for computer vision applications. All together, they are capable of delivering up to 64 integer MACS per clock cycle. This translates to roughly 57 GMACS when running at 900 MHz clock frequency.

Table 6 compares the complexity of the dense form and its corresponding sparsified form for JSegNet21 CNN model. It shows the actual Giga MACS and Giga Cycles measurements from TDA2x SoC for inferring the segmentation of one frame of size 1024x512. Frames per Second (FPS) that can be achieved are also given. Without utilizing sparsity the FPS that can be achieved is 5.14. When using sparse BMA kernels and L1 regularized training, the FPS increases to 12.04. In addition, by inducing a sparsity of 80% the FPS increases to 20.22. The overall speedup is by a factor of 3.93x.

Note that the quantization stage alters the sparsity distribution in layers and even with 80% sparsity induction, the final sparsity achieved after quantization is large in some layers. This alteration is higher in layers with large number of channels where the coefficients tend to be small. Thus the convolution layers (layers 13, 14 in Table 3) with most of the complexity, has a higher sparsity after the quantization than the sparsity that was induced.

Inference method	Configuration for inference	Giga Macs	Giga Cycles	Time (Milli-Seconds)	Frames Per Second
Dense	Without utilizing sparsity	8.843	0.700	194.44	5.14
Sparse	L2 regularized trained	8.163	0.653	181.39	5.51
	L1 regularized trained	3.264	0.299	83.06	12.04
	Sparsity induced at 80%	1.540	0.188	52.22	20.22

Table 6. Measurements from TDA2x SoC for inferring semantic segmentation of an image of 1024x512 resolution

5. Conclusion

The sparsity inducing step is based on thresholding and can be done quickly. The subsequent fine tuning step converges in reasonable number of iterations. This is a significant improvement in speed of thresholding compared to [12]. We also get reasonably precise control on the amount of sparsity induced. Usage of sparsity results in nearly 4x improvement in the speed of inference on TDA2x using 80% sparsity induction.

As noted earlier, CNN inference using 8-bit quantization can be much faster than floating point implementations. The quantization method described in this paper enables inference of CNN using 8-bit fixed point operations. This quality loss observed is minimal.

Optimized implementation on TDA2x SoC shows that the proposed approach is suitable for embedded implementations and remarkable speedup can be obtained with minimal quality loss. Source code and scripts are provided for training sparse and quantized models.

It may be possible to induce even higher sparsity than 80% by the quick sparsification method, without incurring significant quality loss. This is an area for future research.

Figure 3. Sample input images from the validation set and the segmentation produced using sparse (80%), quantized inference (chroma blended visualization)

References

[1] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet classification with deep convolutional neural networks. In NIPS, 2012.

[2] Girshick, R., Donahue, J., Darrell, T., Malik, J.: Rich feature hierarchies for accurate object detection and semantic segmentation. In: CVPR. (2014)

[3] Evan Shelhamer, Jonathan Long, Trevor Darrell, Fully Convolutional Models for Semantic Segmentation PAMI, 2016

[4] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon and Ali Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. ECCV 2016.

[5] Fengfu Li, Bo Zhang, Bin Liu, Ternary Weight Networks, arXiv preprint arXiv:1605.04711

[6] Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun, Deep Residual Learning for Image Recognition, CVPR, 2016

[7] Marcel Simon, Erik Rodner, Joachim Denzler, ImageNet pre-trained models with batch normalization, arXiv preprint, arXiv:1612.01452 [cs.CV]

[8] K. Simonyan, A. Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv technical report, 2014

[9] Murtaza Ali, Eric Stotzer, Francisco D. Igual, Robert A. van de Geijn, Level-3 BLAS on the TI C6678 Multi-core DSP, 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)

[10] TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor, http://www.ti.com/lit/ds/symlink/tms320c6678.pdf

[11] Max Jaderberg, Andrea Vedaldi, Andrew Zisserman. Speeding up Convolutional Neural Networks with Low Rank Expansions, British Machine Vision Conference, 2014

[12] Wei Wen, Chunpeng Wu, Yandan Wang, Learning Structured Sparsity in Deep Neural Networks, NIPS 2016

[13] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. “Hardware-oriented Approximation of Convolutional Neural Networks.“, International Conference on Learning Representation (ICLR) – workshop track, May 2016

[14] Tensorflow. https://www.tensorflow.org/

[15] Pete Warden. TensorFlow: Enabling Mobile and Embedded Machine Intelligence, Emdedded Vision Summit, May 2016.

[16] Jeff Dean, Large-Scale Deep Learning for Intelligent Computer Systems, Emdedded Vision Summit, May 2016.

[17] TDAx ADAS SoCs, http://www.ti.com/lsds/ti/processors/dsp/automotive_processors/tdax_adas_socs/overview.page

[18] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The Cityscapes Dataset for Semantic Urban Scene Understanding," in Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[19] Fisher Yu and Vladlen Koltun. Multi-Scale Context Aggregation by Dilated Convolutions, ICLR 20