XNOR Networks for Embedded Systems

Andrei Cramariuc
Semester Project, Autumn 2016
Integrated Systems Laboratory
crandrei@student.ethz.ch

Supervisor
Prof. Luca Benini
Co-supervisors
Lukas Cavigelli
Renzo Andri

***Abstract*—Convolutional neural networks (CNNs) have risen to be the most commonly used classifiers nowadays in domains such as image and audio processing, for their ability to model complex data. One of the disadvantages of CNNs is the huge computational complexity and the large amount of memory that is required for both training and inference, where most of the workload is usually offloaded onto power-hungry and large Graphical Processing Units (GPUs). On the other hand there exist many applications on low-power embedded systems that would benefit from using state-of-the-art classifiers, but are currently limited by the high energy and computational demands of CNNs. This project explores the possibility of using XNOR networks which are CNN that have been trained to only use binary weights and activations, thus reducing the amount of required memory and speeding up computation significantly. The disadvantage of this quantization is that XNOR networks have on average a 10% lower accuracy. The aim of this project will be to implement, train and test an XNOR network on a microcontroller.**

I. Introduction

Convolutional neural networks have significantly advanced the limits of machine learning in a wide variety of tasks in the domains of image and audio processing. CNNs are currently considered state-of-the-art classifiers when it comes to problems such as object recognition [1], image segmentation [2] and speech recognition [3]. Many modern technologies such as self driving cars, facial detection algorithms and image tagging algorithms rely heavily on CNNs.

Today, due to the high amount of computational power required to train and run CNNs, they are almost exclusively used in conjunction with one or more GPUs [1]. GPUs offer a considerable speed up by calculating in parallel a large amount required computations, but at the cost of a significant increase in energy consumption. Using GPUs also adds the need for dedicated hardware which increases the amount of required space and the cost of production.

Large amounts of memory are required because CNNs have very many floating-point parameters, sometimes even hundreds of millions. For example, a typical CNN structure for image classification such as AlexNet has 61 M parameters that need to be stored and a forward pass requires a total of 1.5 billion high precision operations per image [6]. As a result it is a challenge to run CNNs on low-power devices with limited amounts of computational power, memory and storage. Having a state-of-the-art classifier running on embedded systems would be beneficial in many situations in which processing the data off-chip is not possible or transmitting the data would be more costly than processing it locally. Examples of possible applications include the concept of Internet of Things (IoT), which is gaining popularity due to the rising number of sensors being integrated into our everyday lives, through things such as smart phones, health bracelets and household appliances [4]. Another benefit is the possibility of locally analysing data on existing platforms which satisfies many privacy concerns. Other applications include satellites where data transmission is complicated or mobile robots that would need to be able to operate completely independently [5].

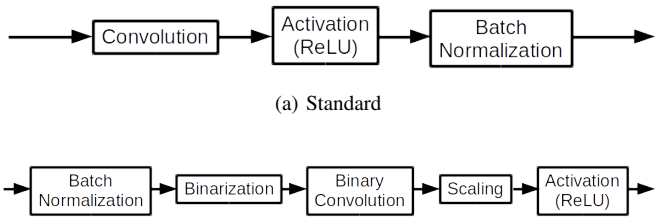
As there is significant interest in enabling the usage of CNNs on embedded platforms different approaches have been proposed. One of these approaches is to simplify the neural network design, by using shallow networks that in theory can approximate any decision boundary but in practice are difficult to train [7]. Another possible simplification is using more compact layers, which can be done for example by replacing the fully connected layers at the top of a CNN with global average pooling [8] or replacing convolutional filters with smaller ones [9]. Another approach to simplify neural networks is compressing them by pruning unused parameters, removing unused connections and quantizing existing weights [10], [11]. A more drastic approach to quantization is binarizing all the parameters in the network during training and inference [12].

This project aims to implement and test the method proposed in Rastegari *et al.* [13]. They binarize both the parameters and the activations of the network, to produce what they call XNOR networks. This binarization greatly reduces the required amount of memory and computational power but introduces an equally big quantization error. Parameters can now be stored using only one bit instead of a 32-bit floating-point and multiplications can be done using bitwise operation, addition and subtraction. Using only binary values it is also possible to perform several multiplications at the same time by storing multiple binary parameters in the same variable. This project aims to explore how this method can be applied to other networks than the one tested in Rastegari *et al.* [13] and also to test the applicability of the method in practice on a microcontroller.

II. Related works

A. XNOR neural networks

Rastegari *et al.* explore the option of binarizing both the weights and the activations of a CNN so that all the values are represented by either 1 or -1 [13]. In practice these values can be represented by only one bit instead of the 32 bits which



(a) Standard

(b) XNOR

Fig. 1. The basic blocks in the structure of standard CNNs and XNOR networks.

are required to represent a floating-point value in a standard CNN layer.

Given a floating-point filter $\mathbf{W} \in \mathbb{R}^{h \times w \times c}$ of height h, width w and with c input channels, the aim is to estimate $\mathbf{W}$ with a binary filter $\mathbf{B} \in \{+1, -1\}^{h \times w \times c}$ using a scaling factor $\alpha \in \mathbb{R}^+$ so that

$$\mathbf{W} \approx \alpha \mathbf{B} \tag{1}$$

To solve $\mathbf{B}$ and α we have an optimization problem that can be written as

$$\underset{\mathbf{B},\alpha}{\operatorname{argmin}} \; ||\mathbf{W} - \alpha \mathbf{B}||^2 \tag{2}$$

for which the solution is

$$\mathbf{B} = \operatorname{sign}(\mathbf{W}) \tag{3}$$

$$\alpha = \frac{1}{n} ||\mathbf{W}||_{l1} \tag{4}$$

where $n = h \cdot w \cdot c$ is the number of elements in $\mathbf{W}$ [13]. Therefore the optimal binary filter is $\mathbf{W}$ thresholded at zero and the optimal scaling factor is the average of the absolute values of $\mathbf{W}$. Also the fully connected layers of a CNN can be binarized using this method. This is because a fully connected layer can be written as a set of filters where there are as many filters as there are neurons on the fully connected layer and all the filters are equal in size to the previous layer. Therefore the above rule is sufficient in binarizing the majority of the parameters of a CNN.

Binarizing the activations $\mathbf{A} \in \mathbb{R}$ is done through adding a threshholding layer equivalent to $\operatorname{sign}(\mathbf{A})$ and slightly reorganizing the network structure to compensate for the error caused by the quantization. A block from which a standard CNN is formed is described in Figure 1a. When binarizing the activations the batch normalization is moved before the binarization. Here the batch normalization layer normalizes the input and applies an affine transform learned during the training phase. The batch normalization layer can be described as

$$\operatorname{BatchNormalize}(\mathbf{A}) = \frac{\mathbf{A} - \mu}{\sigma} \cdot \gamma + \beta \tag{5}$$

where μ is the mean of the batch, σ is the standard deviation of the batch, and γ and β are the parameters of the affine transform learned during training. Situating the batch normalization layer before the binarization makes it so the network tcan learn the optimal scaling factor and threshold for the following binarization layer.

Fig. 2. The MFCCs of a guitar string as a 64×400px image

After binarizing both the activations and the weights of the network, the convolution operation can be done using only bitwise operations, addition and subtraction. This approach avoids the use of multiplication which on many embedded systems is a costly operation without dedicated hardware support. When a Floating-Point Unit (FPU) is not available, multiplication operations between floating-point values are emulated in software using fixed-point operations. Emulated fixed-point operations are slower because they require multiple basic operations, as opposed to a FPU which can perform most operations in a single instruction cycle.

The training of the network still has to be done using full-precision weights, because the gradient update would otherwise be too small to influence the weights. During training the full precision network is binarized and the direction of the gradient is calculated. The stochastic gradient descent algorithm then updates the weights of the full-precision network based on the direction of the gradient. The activations are binarized similarly during training and run time.

Rastegari *et al.* tested XNOR networks on AlexNet trained on the ImageNet classification task [13]. The ImageNet dataset consists of 10 million hand labeled images from over 10 000 categories. The network was trained on a subset of ImageNet containing 1.2 million images from 1000 categories [15]. Nowadays the best results that have been obtained on ImageNet have a mean average precision of 66% [16]. Using full-precision weight and activations the resulting accuracy for the network trained by Rastegari *et al.* was 56.6%. Binarizing the weights did not affect the accuracy, but also binarizing the activations resulted in an accuracy of 44.2%. [13]

B. Audio event classification

To test XNOR networks this project focused on binarizing and implementing the network published by Meyer *et al.* [14]. This network was trained classifying audio events into 28 classes including: cat, laughter, helicopter, engine, footstep, bird, violin and dog sounds. The dataset with which the network was trained consists of a total of 5223 audio samples at 16 kHz and with a varying length of 1 to 20 seconds and a total duration of 768 minutes.

The preprocessing was done using the Mel-frequency cepstral coefficients (MFCCs). The MFCCs of a signal are obtained by taking the Fourier transform of a signal and mapping the logarithms of the powers of the spectrum onto the mel scale. Afterwards the MFCCs are the amplitudes obtained from the discrete cosine transform of the list of mel logarithmic powers. The one-dimensional audio signal is transformed into

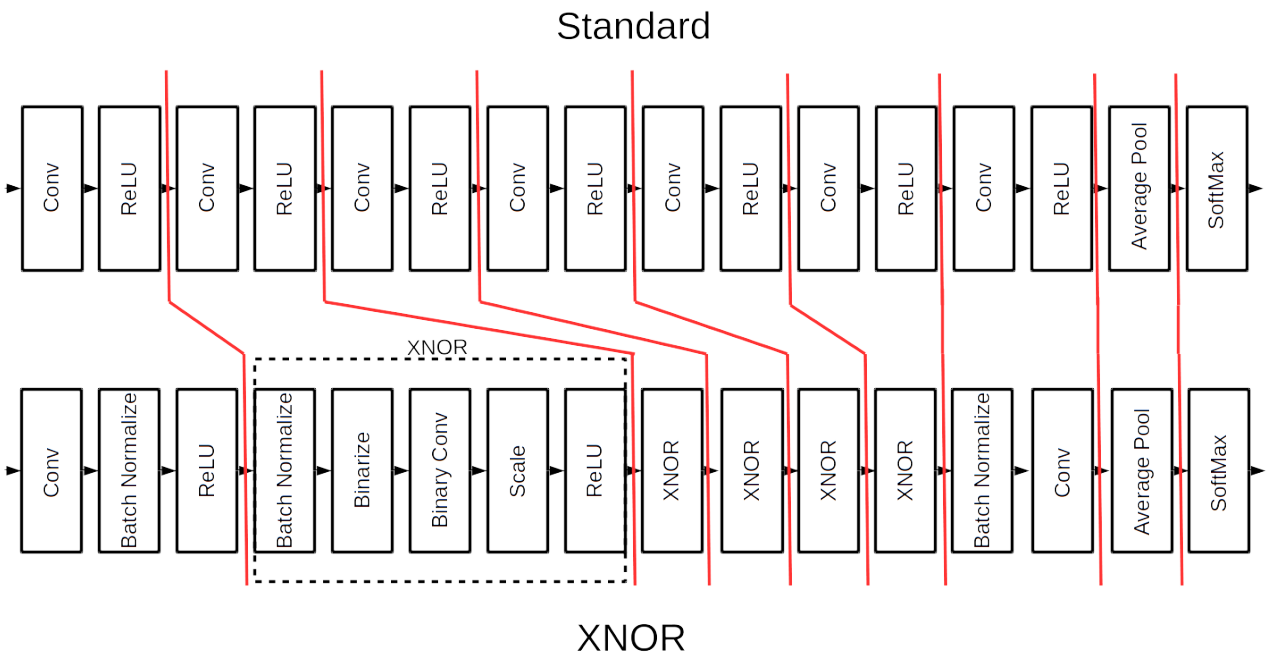


Fig. 3. At the top is the CNN structure proposed by Meyer *et al.* and at the bottom is the modified XNOR network structure [14].

TABLE I
Network structure described by Meyer *et al.* [14].

Layer	Parameters	Memory
conv 3×3, 32	320	1 kB
conv 3×3, 64	18 k	72 kB
conv 3×3, 128	72 k	288 kB
conv 3×3, 128	144 k	576 kB
conv 3×3, 128	144 k	576 kB
conv 1×1, 128	16 k	64 kB
conv 1×1, 28	3.5 k	14 kB
Total:	**395 k**	**1.6 MB**

TABLE II
Number of parameters for each feature map in the network.

Conv layer	Feature map parameters	Memory (full-precision)	Memory (XNOR)
1	25 k	100 kB	100 kB
2	800 k	3200 kB	100 kB
3	400 k	1600 kB	50 kB
4	800 k	3200 kB	100 kB
5	200 k	800 kB	25 kB
6	200 k	800 kB	25 kB
7	200 k	800 kB	-
output	44 k	176 kB	-

a two-dimensional image by windowing the audio signal and taking the 64-point MFCCs of each window and placing them as consecutive columns in an image. Figure 2 shows what the sound of a swinging guitar string looks like. In total the image contains 400 windows each with 400 samples and where consecutive windows overlap 75%. Each window therefore represents 2.5 seconds of audio at 16 kHz. MFCCs are used because they compactly represent the frequencies in sounds similarly to how humans perceive them. MFCCs have been successfully used in modelling both speech and music [17].

The CNN structure proposed by Meyer *et al.* has in total 7 convolutional layers which are listed in Table I [14]. In Table I each row in the first column represents the size of the convolutional filter and the number of filters. The second column is the number of parameters for each layer and the third column is the amount of memory needed when using floating-point units. The structure of the network is in Figure 3. The network has already been compressed by replacing the fully connected layers at the top with smaller convolutional layers and a global average pooling layer. The total number of parameters is 395 k and the memory required to store them is 1.6 MB when using floating-points. The size of the feature map that each convolutional layer takes as input is in Table II and the total amount of required intermediary storage is the sum of the two largest consecutive feature maps, which is 1200 k parameters. When using 32-bit floating-point values the network would require about 1.6 MB of memory for weights and 4.7 MB for feature maps, which is a total of 6.2 MB. The full-precision network achieved a top-1 classification accuracy of 86% and a top-5 classification accuracy of 100% [14].

TABLE III
Number of parameters of the equivalent XNOR network for the CNN described by Meyer *et al.* [14].

Conv layer	Binary	Floating-point	Memory
1	-	448	2 kB
2	18 k	256	3 kB
3	72 k	512	11 kB
4	144 k	768	21 kB
5	144 k	768	21 kB
6	16 k	768	5 kB
7	-	4 k	16 kB
Total:	**394 k**	**7.4 k**	**79 kB**

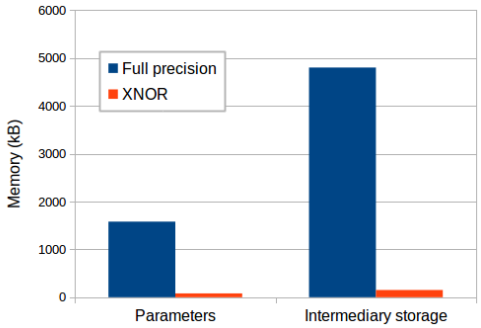


Fig. 4. Diagram of how the data is organized in the feature map of the XNOR network.

III. Implementation

A. Network structure

The network structure proposed by Meyer *et al.* was modified according to the changes proposed by Rastegari *et al.* [13], [14]. The weights for the first and last convolutional layers and their activations were not binarized, because they caused a too significant loss in accuracy [13]. Not binarizing these layers has only a minor impact on the reduction in complexity and required memory, due to the fact that the first and last layer are the smallest in the network as can be seen from Table I. The full-precision intermediary storage required by the last layer can be ignored in practice since the last layer can be combined with the previous layer and the resulting output can be directly accumulated into the global average pooling layer. Therefore the maximum amount of required intermediary storage for the XNOR network is 150 kB.

As opposed to what was suggested in [13] the activation layers were left in the binarized version of the network, because it was empirically determined that keeping them increased the final accuracy. A batch normalization layer was added between the first convolutional layer and proceeding ReLU layer. The activation layer proceeding the last convolutional layer was removed and a batch normalization layer was added in front of the convolutional layer. The intermediary five convolutional layers and their activation layers were replaced with a batch normalization, a binzarization, a binary convolution, a scaling and a ReLU layer. The final structure of the network is in Figure 3 and can be compared with the original network structure in Figure 3.

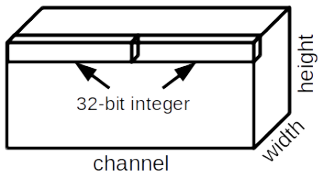


Fig. 5. Diagram of how the data is organized in the feature map of the XNOR network.

The non binary parameters still remaining in the XNOR network structure are the bias and scaling factor of each binary convolutional filter, the batch normalization parameters and the full-precision first and last convolutional layers. Table III contains the total number of full precision and binary parameters, which require a total of 79 kB. The ratio between the number of full-precision parameters and binary parameters remains less than 2%. The total amount of require memory for a XNOR net is 230 kB which is 28 times less than for the original network. A comparison between the required memory for the two networks is in Figure 4.

B. Convolution

For this project the forward pass of the network was implemented in C++ so that it could be run on a microcontroller. One 32-bit unsigned integer was used to store 32 binary parameters at the same time. The values were organized in the following way: consecutive bits in an integer corresponded to values at the same position in consecutive channels. An exemplification can be seen in Figure 5. The data is similarly organized in each filter so that the feature map values in each channel align with the corresponding values of the filter for that channel.

When using XNOR networks the multiplication of two values needed when computing convolutions can be done using

$$filter \cdot input = 1 - 2 \cdot (filter \oplus input) \tag{6}$$

where $filter$ is the value of the filter for a certain position and $input$ is the value of the input for that corresponding position. The multiplication by two in Equation (6) can in practice be implemented as a bitwise shift to the left by 1, which is a less computationally demanding operation than multiplication. Modifying Equation (6) to apply to 32-bit integers is simple and can be done using the function popcount which tells how many bits in a integer are 1.

$$filter_{32} \cdot input_{32} = 32 - 2 \cdot \text{popcount}(filter_{32} \oplus input_{32}) \tag{7}$$

where $filter_{32}$ is 32 values of the binary filter as represented in Figure 5 and $input_{32}$ is the corresponding 32 binary inputs. Therefore using Equation (7) it is possible to do 32 multiplications at the same time. On some architectures popcount is a built-in function and when that is not the case it can be implemented without using branch instructions and only using around 20 instruction cycles or less with lookup tables.

Even when using floating-points with a FPU a minimum of 32 instruction cycles is required for the multiplications. In addition when multiplying values individually a loop, as well as more writes and reads to and from the memory are needed.

C. Consecutive layers

Another advantage of XNOR networks is that intermediary values between two binary convolutions can be manipulated without using floating-points. The intermediary layers between a binary convolution and a previous convolution layer can be simplified to three comparisons. This is because the result of the multiplication and summation of the filter with the input x at one position is biased and scaled and afterwards passed through the batch normalization, ReLU and binarization layers, before going into the next binary convolution layer. All these operations can done in-line and then we would only need to store the binary output of the binarization layer. The operations performed on x before the next convolutional layer can be written as

$$x_{conv} = x \cdot \alpha + bias \tag{8}$$

where $bias$ is the bias of the filter and α is the scaling factor calculated based on Equation (4). Afterwards is the ReLU layer which is equivalent to

$$x_{ReLU} = \max(x_{conv}, 0.0) \tag{9}$$

The scaling and affine transform applied by the batch normalization layer from Equation (5) is equivalent to

$$x_{BatchNormalize} = \frac{x_{ReLU} - \mu}{\sigma} \cdot \gamma + \beta \tag{10}$$

And the final layer is the binarization layer which can be written using sign but it is enough to know if $x_{BatchNormalize}$ is greater than zero or not.

$$x_{BatchNormalize} > 0 \tag{11}$$

Now we want to know what x has to be so that the condition in Equation (11) is satisfied. We start by combining Equations (10) and (11)

$$\frac{x_{ReLU} - \mu}{\sigma} \cdot \gamma + \beta > 0 \tag{12}$$

Since the standard deviation σ is by definition positive it can be moved to the right side of the inequality. The parameter γ which is the scaling factor of the affine transform that is part of the batch normalization layer can be negative so it is only possible to divide by its absolute value. From Equation (12) we obtain

$$\text{sign}(\gamma) \cdot x_{ReLU} > -\frac{\beta\sigma}{|\gamma|} + \text{sign}(\gamma) \cdot \mu \tag{13}$$

For simplicity we denote the right side of the inequality Equation (13) by M_1 which is a floating-point constant. We continue by combining Equations (13), (8) and (9)

$$\text{sign}(\gamma) \cdot \max(x \cdot \alpha + bias, 0.0) > M_1 \tag{14}$$

Fig. 6. The STM32F469I Discovery board used in this project.

TABLE IV
Characteristics of the STM32F469NI Microcontroller.

Inscruction Set	ARMv7-M (32-bit)
FPU	✓
Frequency	180 MHz
SRAM	384 kB
Flash	2 MB
Supply voltage	3.3 V

From (4) we know that α is positive so we can write

$$\text{sign}(\gamma) \cdot \max(x, -\frac{bias}{\alpha}) > \frac{M_1 - sign(\gamma) \cdot bias}{\alpha} \tag{15}$$

Again we can replace the right side of the inequality with a floating-point constant M_2 to obtain

$$\text{sign}(\gamma) \cdot \max(x, -\frac{bias}{\alpha}) > M_2 \tag{16}$$

$$M_2 = (-\frac{\beta\sigma}{|\gamma|} + \text{sign}(\gamma) \cdot (\mu - bias)) \cdot \frac{1}{\alpha} \tag{17}$$

The constants in Equation (16) can be precomputed when the network is initialized which also saves memory since now we only need to store two values and a sign. In practice this can be implemented either by changing sign and doing two comparisons or by doing three comparisons and moving the sign to the right side of the inequality at initialization. When applying this simplification to the intermediary layers between two binary convolutions the added condition is that x is an integer. In this case $-\frac{bias}{\alpha}$ and M_2 can be truncated to integers and we also need to store a variable that tells which one of them is larger for the special case in which they truncate to the same number but as floating-points one is greater than the other. In this way consecutive XNOR layers do not require any floating-point operations nor storing into memory any floating-point values for the XNOR layers or the layers in between them.

D. Platform

The platform on which the network was tested was a STM32F469I Discovery board from STMicroelectronics fea-

turing an ARM M4F microcontroller. A picture of the development board is depicted in Figure 6. The microcontroller has a 32-bit instruction set and contains a hardware FPU. The microcontroller has an operating voltage of 3.3 V and 384 kB of SRAM storage. A summary of the features of the microcontroller is in Table IV. Important when doing convolutions with floating-point values is the number of cycles needed for a Multiply-Accumulate operation (MAC) which with the integrated FPU is 3. Other basic operations such as XOR, addition and subtraction only require 1 instruction cycle. Also to note is that the microcontroller does not have a built-in instruction for popcount, therefore in this case it was implemented in software.

E. Memory optimizations

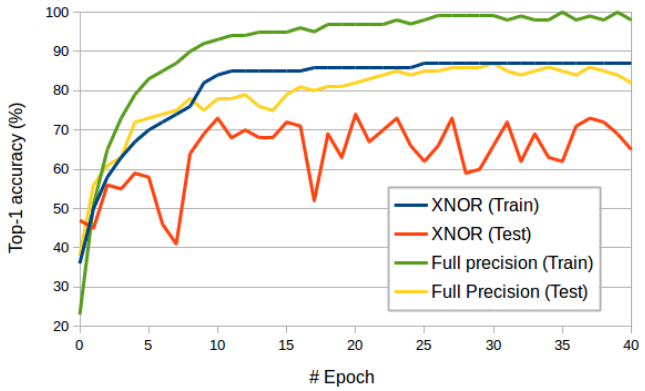
The minimum amount of memory required for the XNOR network is 230 kB based on Tables II and III. Theoretically it would be possible to run the network all at once on the microcontroller, but as a simplification the input image was divided into 4 overlapping section. This simplification allows for a more easier implementation regarding how memory buffers are allocated and used. Each overlapping section is passed separately, memorizing the result and adding all of them together at the end to obtain the final result. The overlap causes a computational overhead of 15% and reduces the memory required for storing intermediary results to a fourth. The final implementation uses a total of 190 kB of memory due to overheads and theoretically simple optimizations such as buffer memory management that were not added.

Another optimization is that the last XNOR network would need to have a floating-point output for the last layer which is full-precision. To not allocate such a large floating-point buffer for only one output the last and second to last layers were combined into one. This is simply done since both convolutional layers only have 1×1 filters in which case it is simple to pass the input through both of the layers at the same time. This makes separate measurements for these two layers impossible and therefore they will be referred to only as one layer in the Results section.

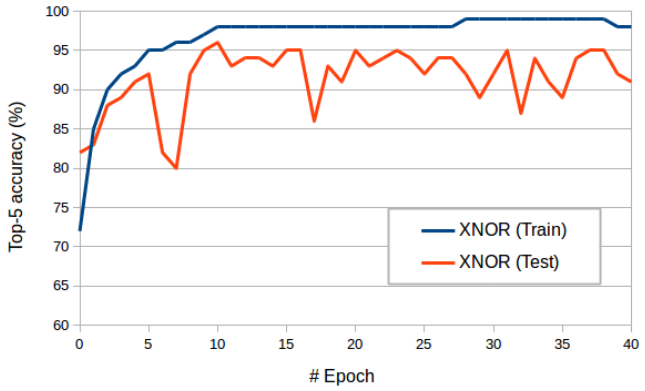
IV. Results

A. Training

The training of the network was done using the framework Torch on a Nvidia GeForce GTX 1080 GPU with 8GB of memory. The training data was the as presented in Meyer *et al.* with the same split of 75% training data and 25% test [14]. The XNOR network was trained using a batch size of 64, which uses about 5GB of GPU memory and takes 2 minutes for each epoch to complete. The training was done for a total of 40 epochs. The optimizer used to train the network was Stochastic Gradient Descent (SGD), with a momentum of 0.9 and with a decaying learning rate. Starting with a learning rate of 10^{-2} at the beginning the XNOR network learns quickly but soon afterwards tends to overfit. To avoid this the learning rate is decrease significantly after the first 8 epochs to 10^{-4}.



(a) Top-1 accuracy for XNOR and full precision during the training phase



(b) Top-5 accuracy for the XNOR network during the training phase

Fig. 7. Top-1 and top-5 accuracies during the training phase for the XNOR network also in comparison with the top-1 accuracy during training for the full precision network.

The accuracy results for the network are visible in Figure 7. The graph in Figure 7a compares the top-1 accuracy of the full precision network with that of the XNOR network. As can be seen the XNOR network reaches a maximum accuracy of 74% while the full precision network reaches an accuracy of 86% which is a difference of 12%. The difference coincides with the results obtained by Rastegari *et al.* where the binarization of the network similarly caused a 12% drop in accuracy [13]. The accuracy of the XNOR network on the training set remains constant but stabilizes at 88%, which is also 12% less than the accuracy, on the training set, of the full precision network which stabilizes at 100%. As seen in Figure 7b, the top-5 accuracy reached by the full precision network is 100% while the XNOR network only manages to reach 95% on the test set.

The classification accuracy of the XNOR network on the test set in Figure 7a is stable on average but has a large variance compared to the full precision network. This instability is because of the binarization of the weights as well as the activations which causes the XNOR network to have trouble fully representing the internal structure of the data. This also makes the XNOR network very sensitive to changes, for

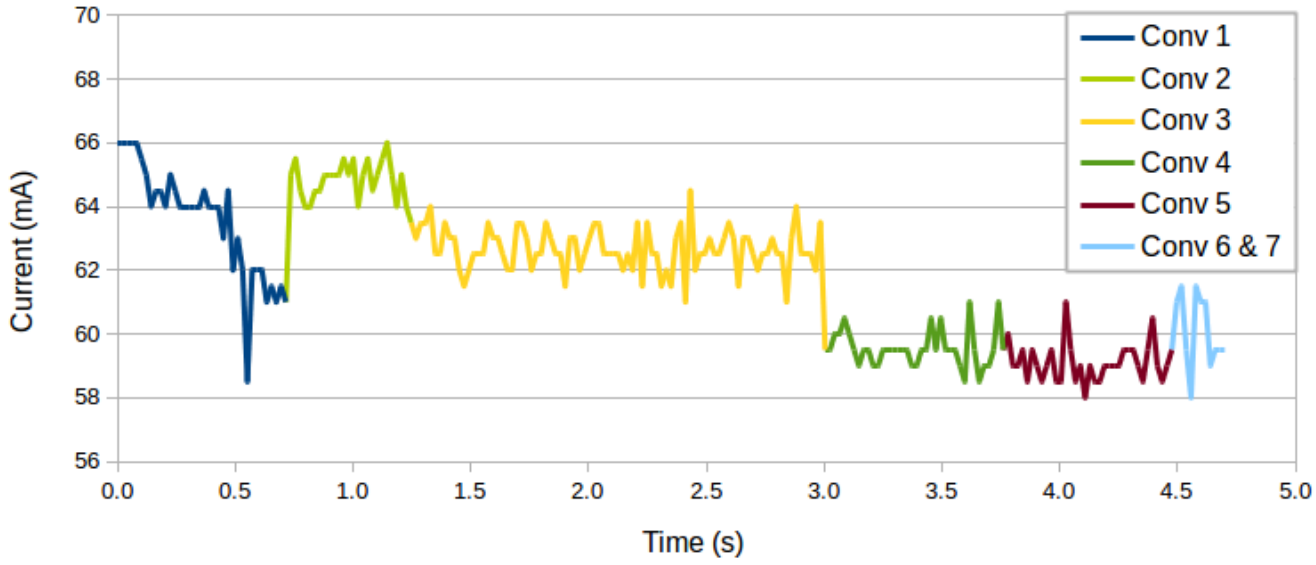


Fig. 8. Current consumption for the entire network

TABLE V
Time necessary to compute each convolutional layer in the XNOR network for the whole input image.

Conv layer	Time (s)	Energy (J)
1	3.2	0.67
2	2.0	0.42
3	6.4	1.31
4	3.0	0.60
5	3.0	0.60
6 & 7	1.2	0.24
Total	**19 s**	**3.8 J**

TABLE VI
Total number of MACs necessary for the full image for each layer of the full precision network taken based on Meyer *et al.* [14].

Conv layer	MACs	Throughput (MAC/s)	Efficiency (MAC/J)
1	7.4 M	2.3 M	11 M
2	118 M	59 M	281 M
3	472 M	74 M	360 M
4	236 M	79 M	393 M
5	236 M	79 M	393 M
6 & 7	32 M	27 M	133 M
Total/Average	**1100 M**	**58 M**	**290 M**

example removing the ReLU layers causes an almost 20% drop in accuracy. Similarly for example removing the first batch normalization layer reduces the average accuracy by 5%. As mentioned in Section III-A the first and last layers are not binarized since this brings only a small benefit causes a significant drop in accuracy. Binarizing the weights and activations for the last layer causes a 10% drop in accuracy and similarly binarizing the first layers weight and activations causes a complete loss in accuracy.

B. Forward pass

The algorithm was tested on the development board described in Section III-D and the current consumed by the microcontroller was measured using a Keysight N6705A power analyzer. The resulting current for each layer was differentiated using a trigger signal. The graph for the current usage of the microcontroller can be seen in Figure 8. Figure 8 only represents passing one fourth of the input through the network, for a full prediction four such cycles are required. Since the microcontroller operates at a voltage of 3.3 V and the time it takes to compute each layer is know from Table V it is possible to calculate the required energy for each layer. Classifying 2.5 seconds of audio therefore takes 19 seconds and requires 3.8 J energy.

The efficiency and throughput of the network can be calculated using Table V and the number of MACs for each convolutional layer calculated by Meyer *et al* [14]. The results are shown in Table VI from where we can see that the total throughput for the XNOR network is 58 M MAC/s and the energy efficiency is 290 M MAC/J. Of interest are also the throughputs for the full precision and XNOR convolutional layers. We will ignore the last row which contains the last two layers of the network combined together since one is a XNOR layer and the other is full precision. We can estimate the throughput and efficiency of a full precision layer to be equal to that of the first layer. Averaging the throughputs of the layers 2 to 5 we get an average throughput of 74 M MAC/s and an average efficiency of 370 M MAC/J. Since it is not possible to test the network using full precision layers on the

TABLE VII
Comparison between a XNOR and a full precision network. Values marked with a (*) were extrapolated from existing measurements.

	XNOR	Full precision
Memory	230 kB	6380 kB
Accuracy (top-1)	74%	86%
Accuracy (top-5)	96%	100%
MACs	1100 M	1100 M
Time	19 s	480 s*
Energy	3.8 J	100 J*
Throughput	58 M MAC/s	2.3 M MAC/s*
Efficincy	290 M MAC/J	11 M MAC/J*

microcontroller we can use the previously calculated values to extrapolate that the full precision network would require 8 minutes and 100J of energy. Which is 25 times slower than the XNOR network and in consequence also consumes 26 times more energy.

V. Conclusions

A final overview of the differences between a XNOR and a full precision network is in Table VII. The XNOR network proved to be more efficient than the full precision network, using 28 times less memory and being 25 times faster and 26 times more energy efficient. On the other hand the XNOR network suffers a loss in precision of 12% when compared to the original network which is consistent with the results obtained also in [13] for another network. When comparing top-5 accuracies the loss is not as significant as the XNOR network has a 96% top-5 accuracy and the full-precision network has a 100% top-5 accuracy. The input image was split into 4 smaller images to simplify the forward pass. This caused the algorithm to be 15% slower than what was theoretically possible but allowed for simplifications regarding the memory management. Therefore the network could be optimized further.

What was seen during training is that even small changes to the structure of the XNOR network had large effects in the accuracy and how easily the network would over-learn. Also the optimizer and its parameters had a very large effect on the outcome of the training. The fact that the accuracy on the training set never reached 100% in combination with the previous observations open the possibility that it could be possible to recoup the loss in accuracy through changes to the network structure and training process. The structure presented in this project is one of many structures that seemed to be the most stable and accurate and there is place for further improvement and the possibility of recouping the loss in accuracy.

The final implementation only used a total of 190 kB memory, which can still be significantly reduced. The use of a FPU was limited to only the first and last layer as well as the initialization of the network. This initialization can be done beforehand and the parameters described in Section III-C can be passed directly to the network. The XNOR network did not completely eliminate the need for floating-point operations, but it reduced the amount to only 1% of the total number of operations. With the reduced amount of floating-point operations needed the XNOR network can also run on a microcontroller that does not have a FPU but uses fixed-point operations.

As a conclusion running simplified version of full precision CNNs on microcontroller and small embedded systems is possible if a loss in accuracy of around 10% is acceptable. In some applications this can be acceptable, for example outdoor sensors or satellites where the cost of transmitting the raw data would be more costly than processing the data locally and only transmitting the result. Using CNNs on embedded platforms would also be useful for cases in which transmitting the data is not desirable for example due to privacy concerns. There exist also systems that can not transmit or need to be able to operate also in cases when transmission fails, such as autonomous robots or self driving cars.

References

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

[2] S.C. Turaga, J.F. Murray, V. Jain *et al.*. Convolutional networks can learn to generate affinity graphs for image segmentation. Neural Computation, 2010.

[3] G. Hinton, L. Deng, G. E. Dahl *et al*. Deep neural networks for acoustic modeling in speech recognition. IEEE Signal Processing Magazine, Nov. 2012.

[4] F. Xia, L.T. Yang, L. Wang *et al.*. Internet of Things. International Journal of Communication Systems, 2012

[5] Y. LeCun, K. Kavukcuoglu, C. Farabet. Convolutional Networks and Applications in Vision. ISCAS, 2010

[6] A. Krizhevsky, I. Sutskever, G.E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. Advances in neural information processing systems, NIPS, 2012

[7] J. Ba R. Caruana. Do Deep Nets Really Need to be Deep?. Advances in Neural Information Processing Systems 27, NIPS, 2014.

[8] C. Szegedy, W. Liu, Y. Jia *et al.*. Going Deeper with Convolutions. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015.

[9] C. Szegedy, S. Ioffe, V. Vanhoucke. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. arXiv preprint arXiv:1602.07261, 2016.

[10] S. Han, J. Pool, J. Tran, W. Dally. Learning both weights and connections for efficient neural network. Advances in Neural Information Processing Systems 28, NIPS, 2015.

[11] S. Han, H. Mao, W.J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. ICLR, 2016.

[12] M. Courbariaux, I. Hubara, D. Soudry *et al.*. Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or -1. arXiv preprint arXiv:1602.02830v3, March 2016.

[13] M. Rastegari, V. Ordonez, J. Redmon *et al.*. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. arXiv preprint arXiv:1603.05279v4, Aug 2016

[14] M. Meyer, L. Cavigelli, L. Thiele. Efficient convolutional neural network for audio event detection. ICASSP, 2017

[15] ImageNet 2012 dataset http://www.image-net.org/challenges/LSVRC/2012/

[16] Results for ImageNet 2016 http://image-net.org/challenges/LSVRC/2016/results

[17] B. Logan. Mel Frequency Cepstral Coefficients for Music Modeling. ISMIR, 2000