ASIC implementation exploration for EfficientNet optimizations

Albin Davidsson
al4767da-s@student.lu.se
Filip Fondelius
fi6320fo-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Academic supervisor:
Liang Liu
liang.liu@eit.lth.se
Ilayda Yaman
ilayda.yaman@eit.lth.se

Supervisor: Anders Lloyd, Axis Communication
Santhosh Nadig, Axis Communication

Examiner: Erik Larsson
erik.larsson@eit.lth.se

July 1, 2024

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Neural networks are effective for solving many complex human-level tasks, one type of neural network often used in imaging tasks is the convolutional neural network (CNN). They can effectively solve problems in areas like image recognition, object detection, and classification tasks. Using a CNN for one of these tasks requires a huge computational cost relative to what a typical algorithm would have used. However, while the processors running the CNNs can handle the computational load, they get extremely hot and draw a lot of power.

This thesis explores different hardware techniques to optimize a fixed convolutional neural network to decrease its power consumption while keeping an acceptable output result. One technique used is precision scaling where the precision of weights and tensors is reduced to a lower number of bits. It reduces both the amount of bits used in calculations and data transfer. Another technique used was approximate computing. These approximate circuits are designed to have fewer gates and therefore draw less power.

In this thesis an algorithm and a simulator was created to choose at which location bits should be removed. By introducing several error metrics such as mean squared error and F1 score, the results could be compared between different metrics and an almost optimal solution could be found.

With our precision scaling method, we have been able to save almost half the energy while maintaining an acceptable F1 score of 0.8. When evaluating approximate multipliers for the same task, the results were more difficult to interpret. The approximate multipliers work well in some cases while they work significantly worse in other cases.

Our results show that using hardware techniques like precision scaling and approximate circuits can be used to reduce power consumption.

Popular science summary

In recent years AI has become a powerful tool for solving complex tasks and there are lots of different use cases for it. This thesis uses a convolutional neural network that takes an input image and outputs a class for each pixel in the image. Each pixel is divided into one of three classes, based on which class it is the most likely to belong to. This is called semantic segmentation. The problem with using neural networks are that they are computational heavy which leads to a high power consumption.

There are many ways to reduce the power consumption for neural networks. This thesis focuses on precision scaling where the normally 8-bit long weights and input tensors are reduced to fewer bits. Since fewer bits have to be both transported to the memory as well as used in computations, less power will be consumed. Lowering the precision of the weights and input tensors affects the accuracy of the result. We have developed a method that non-uniformly reduces the bit precision while maintaining an acceptable result. Our method was evaluated with a simulation program, written in Python and C++, which was also developed during this thesis.

The thesis also discusses the possibility of replacing the multiplier in parts of the neural network with approximate multipliers. Approximate multipliers are circuits that behave like normal multipliers but consumes less power at the cost none exact outputs. There are many different approximate multipliers with different levels of error and power consumption.

Using only precision scaling we have been able to cut the power consumption in half.

Acknowledgments

We would like to thank Axis Communications for the opportunity for this exciting master thesis. Especially our supervisors at Axis, Anders Lloyd and Santhosh Nadig who inspired and helped us through challenging times. An extra thanks to all the other people at Axis who helped us as well.

We would also like to thank our supervisors from LTH, Liang Liu and Ilayda Yaman. Your support is appreciated and your help was always close at hand.

Abbreviations

CNN	Convolutional Neural Network
DNN	Deep Neural Network
MBConv	Inverted Residual Block
ReLU	Rectified Linear Unit
ASIC	Application-Specific Integrated Circuit
ALU	Arithmetic Logic Unit
MAE	Mean Absolute Error
MSE	Mean Squared Error
TP	True Positive
FP	False Positive
FN	False Negative
MAC	Multiplier Accumulate
RGB	Red, Green and Blue

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

With the explosion of neural networks in the last few years, new use cases are being discovered rapidly. The neural network used in this thesis is a U-net which is a type of convolutional neural network. The U-net specializes in finding patterns in images to segmenting them into classes. Our U-net consist of seven MBConvs [1] trained to perform image segmentation tasks on the Oxford-IIIT Pet Dataset. Neural networks have often been improved by increasing their size but this comes with the drawback of increasing their already high power consumption. If our network is used on a portable device or a device with limited power access, power consumption must be kept low.

The power consumption is dependent of the amount of bits transported within the chip and to and from external memories as well as the amount of bits used within the calculations. The network used in this thesis is already quantized using TensorFlow so that all weights and input tensors are 8 bit long. This thesis explores the possibility of decreasing the precision for the input tensors and pre-trained weights to fewer bits in order to reduce the power consumption even further. According to [2] this is possible and reducing the precision may still provide good results. However, the best way to reduce precision is not to lower the bit precision across all stages. Instead, it is better to find which stages in the network that can tolerate reduced precision with minimal impact.

To accomplish this, two different algorithms for finding the best solutions have been created, as well as several error metrics to evaluate these results. The error metrics are for both the classified and unclassified output image. Comparing the different outcomes proved harder than expected, as well as the task to determine what a good result actually is. These error metrics are also important to be able to explain the result. The thesis also provides a light introduction to the use of approximate multipliers in an attempt to further reduce the power consumption.

1.1 Goal of the Thesis

The thesis aims to find a way to reduce the power consumption of the U-net without changing its architecture, weights or number of operations. This can be done in different ways and this thesis will mostly investigate precision scaling but also the use of approximate multipliers. To find the optimal bit precision in each weight and input tensor for the lowest power consumption whilst retaining a high accuracy is infeasible.
Since there are 21 layers where both the precision of the weights and input tensors can be reduced from 8 to 1 bit, a total of $8^{42} = 8.5 \times 10^{37}$ different combinations has to be evaluated. To loop through all of these are unrealistic, so an algorithm to reduce the number of bits without having to check every combination had to be created.

Multiple error metrics also had to be formulated to be able to evaluate and compare the results to determine if results are good and how much the precision scaling and approximate multipliers actually affects the output of the network.

Chapter 2

Background

This section gives a brief introduction to how convolutional neural networks work, as well as how the U-net used in this thesis is built. It also explains power consumption and how it might be lowered using quantization and approximate circuits.

2.1 Convolutional Neural Network

Recently the increased capacity in computing has enabled deep neural networks (DNN) to significantly exceed human capacity when it comes to solving certain types of tasks and are therefore widely used.

The convolution neural network (CNN) is a type of neural network that uses convolutions to explore spatial adjacency. This is implemented by applying a convolution between the input image and a kernel which consists of weights. In contrast to a fully connected network, CNNs use weight sharing. This means that the number of connected neurons between the layers becomes much fewer, which leads to fewer parameters. Because of spatial adjacency, CNNs are great at extracting features and learning to recognize patterns from images which makes them suited for tasks such as image classification, image segmentation, and detection.

Convolutional neural networks can vary in size. Some networks consist of 50 layers [3] while the network used in this thesis consists of 7 layers. The output from each layer consists of many two-dimensional feature maps, each containing a different feature. The dimensionality in the feature dimension often becomes larger the deeper the neural networks go. This is because the network has extracted more features from the input image. On the contrary, the resolution of the feature maps often decreases. This helps by reducing the computational cost and extracting more high-level features from the input image. Early layers typically capture lower-level features such as textures and edges, while deeper layers capture more complex features like object parts and shapes.

2.2 The Convolution Operation

Convolution is an operation defined as the integral of one function multiplied by another after one is reflected about the y-axis and shifted [4].

$$s(t) = (x * w)(t) = \int x(t-a)w(a)\,da$$

In the CNN used in this thesis, the convolution can't be continuous because it is represented by digital values and not analog, so a discrete convolution is used.

$$s(t) = (x * w)(n) = \sum_{a=-\infty}^{\infty} x(n-a)w(a)$$

Where w is the weight tensor and x is the input tensor. Convolution is often used at more than one axis. This is because the input images are two-dimensional and therefore the kernels too.

$$S(r,c) = (X * W)(r,c) = \sum_{i_n}\sum_{j_n} X(r-i_n, c-j_n)W(i_n, j_n)$$

The U-net uses a kernel size of 3x3 and padding when doing the convolution. Padding adds extra pixels around the input image or feature map. This enables the kernel to be centralized and can go outside the actual image to get values which mean that the spatial dimension will be reserved. So the coordinate $X(r,c)$ is placed at the kernel coordinate $W(1,1)$.

$$S(r,c) = (X * W)(r,c) = \sum_{i_n}\sum_{j_n} X(r+i_n-1, c+j_n-1)W(i_n, j_n)$$

To use the discrete convolution formula as explained above in a CNN, it iterates through every pixel for all channels and all output channels. Time complexity can be expressed as $k^2 \times r \times c \times d_i \times d_o$ where k is the kernel size, r and c are the numbers of rows and columns. d_i and d_o are the depths for the input tensor and output tensor. The output tensor can be expressed as the below formula.

$$Output = \sum_{d_{o,n}}\sum_{d_{i,n}}\sum_{r_n}\sum_{c_n} S(r,c)$$

A method used to help convolutional neural networks control the area size of the feature map is called stride. It determines how many pixels the kernels skip when it moves across the input tensor, both from left to right and from top to bottom.

2.2.1 Depthwise Convolution

Depthwise convolution is a type of convolution where the channels are convoluted with different kernels. As explained in section 2.2, in traditional CNNs the number of kernels is equal to the number of input channels multiplied by the number of output channels. But in depthwise convolution, the number of kernels is only equal

to the number of input channels which greatly reduces the number of parameters and operations. The time complexity of depthwise convolution then becomes $k^2 \times r \times cdi$ instead of $k^2 \cot r \times c \times di \times do$. Note that the number of input channels is the same as the output channels in depthwise convolution. The total weight size then becomes $k^2 \times d$. So for example, if the kernel size is 3x3 with a depth of 50, it has 450 parameters [1].

2.2.2 Pointwise Convolution

Pointwise convolution is a type of convolution where a 1x1 kernel iterates through every single point. It can be used to reduce or increase the depth of the feature dimension. The reduction in depth can be beneficial in cases where a computation-heavy step follows, for example, a deep 3x3 or 5x5 convolution. This enables the following convolution step to reduce its filter dimension which will save computation time. Besides being used as reductions, they also include the use of acting like a linear bottleneck, which is used in the MBConv [1] [5].

2.3 Network Design

The U-net consists of seven layers where each layer consists of a MBConv [1] block. The MBConv block is an architecture of three stages and is often used for imaging tasks because of its ability to reduce the amount of computations in a network significantly increasing its efficiency. The network uses the combination of depthwise separable convolution, bottleneck layer, and skip connections to create efficient feature extraction. The efficient feature extraction enables the network to have both fewer parameters and computations compared to a traditional convolutional network which will both increase speed and reduce its power consumption.

The network in this thesis is trained to classify each pixel in an input image as one of three classes: heart, background, or boundary. It is trained on the Oxford-IIIT Pet Dataset [6] which contains images of cats and dogs. The heart class should contain all the pixels that are part of the animal, the background class should contain all the pixels part of the background and the boundary class should contain all the pixels in the transition between the animal and the background.

2.3.1 Architecture

The architecture of the network used is a modified U-net which can be seen in Figure 2.1. The network uses both MBConvs and compound scaling. U-nets are commonly used in image segmentation tasks because of their good performance. A U-net consists of two major parts, an encoder and a decoder. The encoder usually decreases the resolution using maxpooling. This network instead decreases the resolution of the image in each stage using a stride of two in the depthwise convolution stage of the MBConv. In the expand stage it also increases the number of channels to extract as many features as possible. To output a probability map, the tensor has to be scaled up to the resolution of the input image. The number of result channels also has to be the same as the number of classes. This is done

in the decoder part of the network which is made up of the same number of steps as the encoder. The decoder consists of three MBConvs which upsample the tensor in between the expand and depthwise convolutions stages using the nearest neighbor. The tensors from all the encoding steps are saved and concatenated into the decoder step with the same resolution to preserve more features, these are called skip connections.

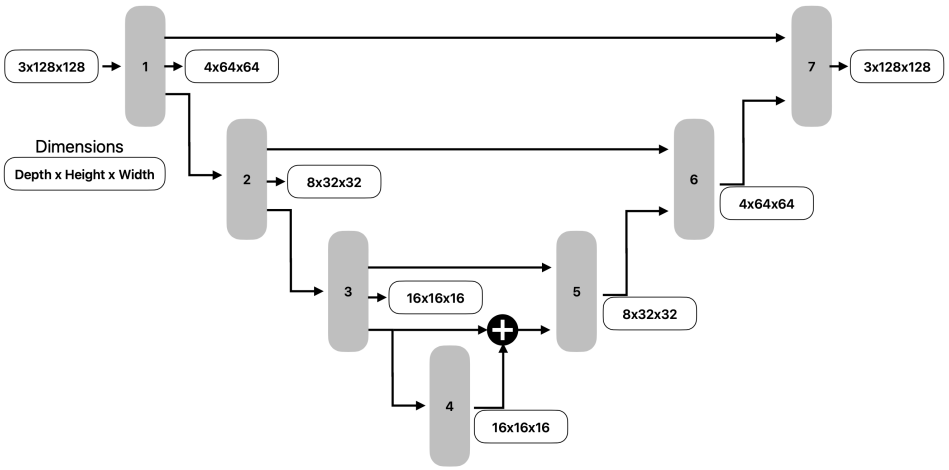


Figure 2.1: The U-net used

2.3.2 Batch Normalization and Activation Function

Batch normalization is a method used both under training and inference to help neural networks get faster and more stable. At inference, this works by applying normalization on the outputs of a stage by re-scaling and re-centering the data, which is the same as multiplying and adding. This means batch normalization is a linear function and does not remove information from our network. The batch normalization during inference in the U-net is different for each layer. In layers 1 to 3, FusedBatchNorm [7] is used. Here is the batch normalization fused into the convolution itself and does not require the operations after the convolution itself. It works by combining the operations for batch normalizing and convolutions into a single kernel.

An activation function is a crucial component in neural networks, used to introduce non-linearity into the network. This enables the network to learn more complex patterns beyond linear ones. An activation function often serves to threshold the output value, depending on its value. In the U-net, the activation function ReLU is used. ReLU stands for rectified linear unit and is defined as the positive part of its argument. It can be described like this:

$$\mathrm{f}\left(x\right) = \max\left(0, x\right)$$

2.3.3 MBConv Block Design

A MBConv Block, also called an inverted residual block, is a type of architecture block for convolutional neural networks. The architecture typically consists of three stages, expansion, depthwise convolution, and output stage. The expansion stage is used for increasing the depth of the tensor by using convolution with a 1x1 kernel. A tensor with more channels allows the depthwise convolutional layer to operate on a richer space of information which increases performance. The expansion stage always applies ReLU as an activation function.

As seen in Figure 2.2, the next stage is depthwise convolution. It performs depthwise convolution on the output tensor from the expand stage with a 3x3 kernel. As said before, a tensor with more channels helps the depthwise convolution extract more features from the input image and the feature map from the previous stage. This stage always applies ReLU to its output as an activation function as well.

The last part of the MBConv in Figure 2.2 is the output stage. This stage reduces the number of channels in the feature dimension by applying pointwise convolution. It serves to reduce the number of channels in the tensor to align the output with the next stage in the network. It also helps maintain computational efficiency. Reducing the number of feature maps also means that there are fewer external memory transfers to the next block. No ReLU is applied here, but rather a linear activation function is applied which helps keep the information precise for the next stage.

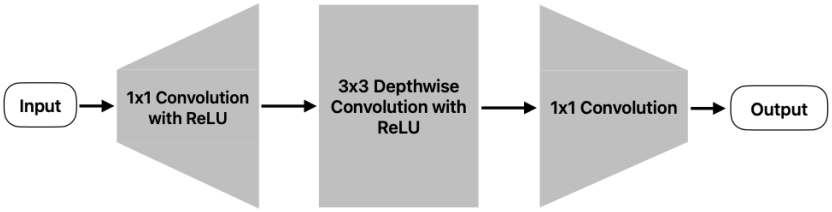


Figure 2.2: Basic MBConv

When the network is implemented, these blocks look a little different depending on which layer they are in. This is because the network first decreases the resolution and then increases it. This is done to get back the original resolution to be able to output the results as a probability map. There are three types of different blocks, which are used in layers 1 to 3, 4, and 5 to 7.

From layers 1 to 3 in Figure 2.1, the MBConv in Figure 2.3 is used. These three steps are what's called an encoder. It reduces the resolution but increases the number of feature channels. Another way to see it is that the image decreases in resolution but the network gains more information about the image and gains additional feature maps.

As seen in Figure 2.3, the expand stage has a variable called $EXPAND$. It's set to 6, which means the expansion stage will increase the feature map dimension by a factor of 6. The expansion stage always applies ReLU to its output as an activation function.

The depthwise convolution in Figure 2.3 also applies stride. By setting it to 2, the area of the feature maps will be reduced by three-quarters. By progressively reducing the area of the feature maps, the network can learn hierarchical features. Early layers focus on low-level features (like edges and textures), while deeper layers capture complex high-level features (like shapes and parts of objects). By reducing the area of the feature map, using a stride of two helps in decreasing the number of multiplication and addition operations needed during the convolution process. This reduction leads to faster computations and less memory usage.

One thing that sets this type of MBConv apart from the others are MUL and ADD in Figure 2.3. By combining these into $f(x) = b + x \times a$, where a is the MUL block, b is the ADD block and x is output from the stage before. The MUL and ADD blocks implement batch normalization after the depthwise convolution stage instead of using FusedBatchNorm [7]. After ADD has been done, ReLu is applied to its output as an activation function.

The output stage in Figure 2.3 applies normal pointwise convolution as explained for the typical MBConv in Figure 2.2.

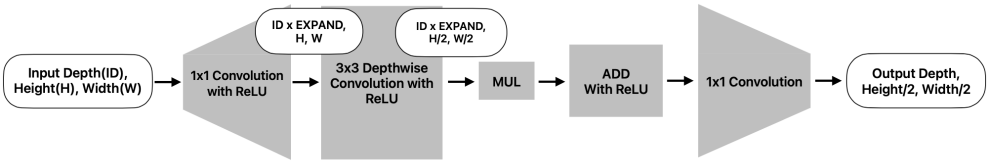


Figure 2.3: MBConv in Layer 1-3

Figure 2.4 details how the MBConv in layer 4 in Figure 2.1 is built. This type of MBConv is the one most similar to the original one in Figure 2.2. The variable $EXPAND$ is set to 6 here and ReLU is applied after the expand and depthwise stage, with no ReLU after the output stage. The depthwise stage keeps the area of the feature maps the same since the stride is set to one.

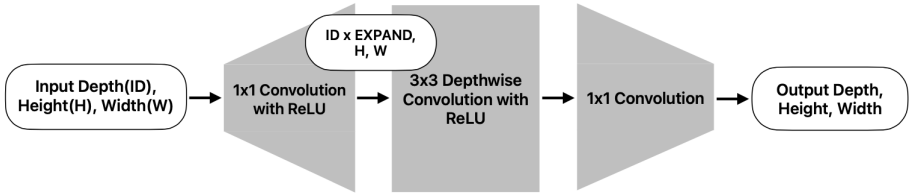


Figure 2.4: MBConv in Layer 4

From layer 5-7 in Figure 2.1, the MBConv in Figure 2.5 is used. These 3 layers are what's called a decoder. It increases the area of the feature maps but reduces the number of feature maps. The decoder collects information from the different feature maps and tries to visualize the information on fewer but bigger feature maps. It goes from deep but small feature maps with the size of (16,16,16) to only 3 different feature maps with the original spatial dimension of the input image to layer 1, which has the tensor size dimension of (3,128,128). The goal of the decoder is to show the result from the encoder on a prediction mask with 3 feature maps, heart, background, and boundary.

As Figure 2.5 shows, the expand stage is the same as for Figure 2.3, and Figure 2.4, with the $EXPAND$ factor set to 6.

The thing that differs from the other MBConvs is the upsampling block. It performs upsampling on the spatial dimension, which is the same as the area of the feature maps. As explained before, this is the goal of the decoder. The upsampling method used is nearest neighbor and the upsampling factor is set to 2. Increasing the spatial dimension and the amount of feature maps also increases the computational cost for these stages.

The depthwise stage and the expansion stage are the same here as for the typical MBConv in Figure 2.2. The stride is set to 1 and depthwise applies ReLU after its convolution while the output stage does not apply ReLU.

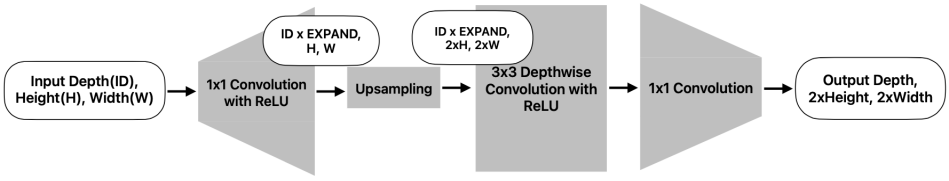


Figure 2.5: MBConv in Layer 5-7

2.4 Power Consumption

Power consumption in ASICs can be divided into two categories, static and dynamic power consumption. Static power consumption, also called leakage power comes from the leakage in the transistors. The static power consumption of a chip, depending on the architecture of the chip, is nowadays usually a lot lower than the dynamic power consumption. The static power consumption can mostly be reduced by choosing transistors that have a lower leakage current. However, these transistors are usually slower and fit better for designs where low power consumption is more important than performance. Dynamic power consumption which is also known as switching power occurs when the transistors switches i.e. goes from 1 to 0 or vice versa. The formula for dynamic power consumption is

$$P_{dyn} = C_L \times f \times V^2 \times A$$

, where C_L is the load capacitance, f is the clock frequency, V is the supply voltage and A is the switching activity [8]. There are several ways to reduce dynamic power consumption. For example, lowering the supply voltage or the clock frequency or using a technology that has a lower capacitance. The design of the circuit also impacts the dynamic power consumption a lot. Here several techniques can be used, both techniques that reduce the amount of calculations and the amount of switching done, like quantization, approximate calculations, skipping, and memorization. This thesis is focused on quantization and approximate calculations and won't investigate skipping or memorization.

2.4.1 Calculation & Memory

The energy consumption for inference can be split into three different stages, the energy required for reading from the memory, the energy required for performing the calculations, and the energy to write back to the memory. The relationship between the energy consumption of the data transport and the energy in the calculation stage is dependent on the chip architecture and the system design.

The energy consumption per multiplication is assumed to be relative to the number of bits that go into the multiplier, since not using the most significant bits in the multiplier will decrease the number of switches for that operation. In this case, all the operations are originally 8-bit multiplied by 8-bits. If the number of bits in the tensor still is 8 but the number of bits in the weight is reduced to 7 the operation only consumes 87.5% of the original energy since $\frac{8\times7}{64} = 0.875$ according to our model for energy estimation explained in Section 3.4.

The memory is where all the short-term data is stored during the execution of the program. Data can be stored in the internal memory, which means it's located close to the ALU. Data can also be stored in an external memory, off-chip. Energy consumed when data is either written or read from internal memory is less than when performing the same operation from the external memory.

2.5 Number of Operations

When considering the U-net with 8-bit precision, the number of MAC operations and the number of internal and external memory transfers are fixed. This information gives an observation of where the energy is being consumed and which type of operation it is. The relative ratios between the three types of operation in different layers and stages can be useful to know where bits are being removed. Another reason why it's important could be the approximate computing elements and where to implement them in the neural network.

Table 2.1, 2.2, and 2.3 shows the number of operations represented by MAC, the number of reads and writes for the internal memory, and the number of reads and writes for the external memory for in each layer. The relative difference between MAC, internal and external memory is also shown, in column 5. The memory number includes both the weight tensors and the input tensors.

Because of the interconnections in Figure 2.1, it can be seen in Table 2.1 that the external memory also increases twice for layers 5-7 if compared to layers 2-4. Layer 1 has the highest number of external memory operations because the original input image dimensions are (3,128,128) and the output of layer 1 (expand stage in layer 2) dimensions are (4,64,64). This gives a relative ratio of 3 between the sizes of these tensors. This difference can be seen between the number of internal memory operations in layer 1 and layer 2. The only difference is that it's not exactly a ratio of 3 because of the added weight tensor memory transfer. The interconnections don't increase the internal memory operations for layers 5-7 as they did for external memory operations. This is because the internal memory operations in the expand stage equal weight tensors read from memory and the output written to memory. None of these are being affected by the interconnections.

Layer in Network	MAC	Internal Memory	External Memory	Ratio (MAC:INT:EXT)
Layer 1	884 736	294 966	49 152	18:6:1
Layer 2	393 216	98 400	16 384	24:6:1
Layer 3	393 216	49 536	8 192	48:6:1
Layer 4	393 216	26 112	4 096	96:6:1
Layer 5	786 432	27 648	8 192	96:3:1
Layer 6	786 432	49 920	16 384	48:3:1
Layer 7	786 432	98 496	32 768	24:3:1
Total	4 423 680	645 078	135 168	33:5:1

Table 2.1: Number of operations for different types in the Expand Stage

The number of operations in Table 2.2 differ from Tables 2.1 and 2.3 at the number of external memory operations. The external memory only transfers data

between the different MBConvs and the depthwise convolution operates inside the MBConv between the expand and output stage. The number of MAC operations in layers 6 and 7 accounts for almost 72% of all MAC operations in the depthwise stage, which means 72% of all consumed energy in the depthwise stages ALU is consumed in layers 6-7. If approximate computing were to be implemented on a large scale, this could be the place for it because computations will dominate the power consumption.

Layer in Network	MAC	Internal Memory	External Memory	Ratio (MAC:INT)
Layer 1	663 552	368 802	0	2:1
Layer 2	221 184	123 096	0	2:1
Layer 3	110 592	61 872	0	2:1
Layer 4	221 184	50 016	0	4:1
Layer 5	884 736	197 472	0	4:1
Layer 6	1 769 472	393 648	0	4:1
Layer 7	3 538 944	786 648	0	4:1
Total	7 409 664	1 981 554	0	4:1

Table 2.2: Number of operations for different types in the Depthwise Stage

Table 2.3 shows a similar number as Table 2.1, but with a heavy increase in the number of operations at the end instead of the beginning. This could be the effect of the decoder and its reversed functionality if compared to the encoder. The expand stage in the encoder will compute a bigger feature map area than the output stage because of the stride in depthwise convolution. However, in the decoder, upsampling is used to magnify the feature map area instead of decreasing it with stride. This is also a reason why all MBConv blocks can't be treated as the same block when it comes to quantization and the reason different quantization needs to apply to them.

Layer in Network	MAC	Internal Memory	External Memory	Ratio (MAC:INT:EXT)
Layer 1	294 912	73 800	16 384	18:5:1
Layer 2	196 608	24 768	8 192	24:3:1
Layer 3	196 608	13 056	4 096	48:3:1
Layer 4	393 216	26 112	4 096	96:6:1
Layer 5	786 432	99 072	8 192	96:12:1
Layer 6	786 432	196 800	16 384	48:12:1
Layer 7	1 179 648	393 288	49 152	24:8:1
Total	3 833 856	826 896	106 496	36:8:1

Table 2.3: Number of operations for different types in the Output Stage

A summary of the amount of operations is shown in Figure 2.4. It can be seen that depthwise convolution almost uses 50% of all MAC operations. Since the quantization of input tensors affects both external memory transfer and MAC operations, it decreases the energy consumed in both operations. But since the depthwise stage does not affect external memory, maybe approximate computing could result in a better result there. This is because approximate computing doesn't reduce external memory operations, but rather reduces the power consumed by the MACs.

Stage in MBConv	MAC	Internal Memory	External Memory	Ratio (MAC:INT:EXT)
Expand	4 423 680	645 078	135 168	33:5:1
Depthwise	7 409 664	1 981 554	0	4:1:-
Output	3 833 856	826 896	106 496	36:8:1
Total	15 667 200	3 453 528	241 664	65:14:1

Table 2.4: Number of operations for different types of convolutions in the whole Network

2.6 Precision Scaling

Precision scaling is a technique to reduce the number of bits used to represent a number. When modern neural networks grow large they require an immense amount of computation and data transport. One way to combat this problem is to reduce the bits used in computation and data transport.

2.6.1 Quantization from Floating-point

Floating-point weights are often used when training a convolutional neural network. To quantize this to integer numbers, abs max quantization or zero-point quantization [9] [10] can be used. When converting from floating-point to signed 8-bit integer, absmax quantization multiplies the number by 127 and then divides it by the max absolute value in the tensor that is being quantized.

$$\boldsymbol{X}_{quant} = \text{round}\left(\frac{127 \times \boldsymbol{X}}{\max\left(|\boldsymbol{X}|\right)}\right)$$

Zero-point quantization is a non-linear method that works differently in the scaling part and adds a zero-point to the formula. This is good for asymmetric inputs which happens after ReLU has been used, because of the only positive number output.

$$\text{scale} = \frac{255}{\max\left(\boldsymbol{X}\right) - \min\left(\boldsymbol{X}\right)}$$

$$\text{zeropoint} = -\text{round}\left(\text{scale} \times \min\left(\boldsymbol{X}\right)\right) - 128$$

$$\boldsymbol{X}_{quant} = \text{round}\left(\text{scale} \times \boldsymbol{X} + \text{zeropoint}\right)$$

2.6.2 Quantization from 8-bit Integer

An 8-bit integer value is in either the range 0 to 255 or -128 to 127 and can be every value in between. Normally when reducing the bits in a value the most significant bit is removed, this comes with the drawback of a reduced range for the value, for example, the range of an unsigned integer goes from the original range of 0 to 255 to 0 to 127. In this thesis, the least significant bit was removed to reduce the precision of the values. This allows the value to keep most of its range but it becomes limited to only using some numbers in the range. For example, a 7-bit unsigned integer has the range 0 to 254 but can only have the even numbers in between.

The model received in this thesis was already quantized from floating-point to 8-bit signed integers. Because of wanting the middle point of the weights to be zero, the weights range from -127 to 127 instead of -128 to 127. To quantize from eight to fewer bits, arithmetic right shift is used combined with rounding to the nearest value expressible using the new number of bits. This makes it perform fast in hardware and easy to implement

2.7 Approximate Multipliers

Another way to reduce the energy consumption of a multiplication is to use an approximate multiplier [11]. These are processing elements that provides an incorrect result for some inputs. They are usually designed to consume less power and most have a shorter critical path.. The reason for this is that an accurate multiplier consists of a certain number of transistors, while approximate components may introduce errors they often require fewer transistors. There are a large amount of variations of approximate multipliers that are built using different amounts of transistors. Different multipliers produce different errors in different intervals, making them useful for different tasks. For example, a multiplier that has fewer errors for low input values will perform better for low input values than for high.

Chapter 3
Method

This section presents how our program is built and introduces the different error metrics and methods to evaluate the results. It also explains how our different algorithms for finding an optimal solution work.

To be able to evaluate different multipliers and bit precisions, a program that performs inference of an input image had to be built. The main part of this program is written in Python, while the operation-heavy convolutions and quantization are written in C++ using pybind. This lets us call our functions written in C++ from our Python code. Doing the convolutions in C++ not only speeds up the program but also lets us change the normal accurate multipliers to hardcoded approximate multipliers. The C++ program includes several callable functions: pointwise convolution, depthwise convolution, mul, add, and two different quantization functions. The difference between the quantization functions is that one takes two dimensional inputs and one takes three dimensional inputs.

The pointwise convolution function performs pointwise convolutions for each depth in the filter. Then it moves to the next column and then the next row. The depthwise convolution function performs a depthwise convolution for each layer before it moves on to the next column and row. The mul and add functions together perform the batch normalization used in the MBConvs in layers 1, 2, and 3. The two quantization functions round every value in the weights and tensors to the nearest value expressible using a specifiable number of bits. In Python, the MBConvs are constructed by calling the C-functions, and the ReLU function is also performed. This approach makes it easy to change the network and try different multipliers and bit precision while it reduces the run time significantly.

3.1 Data set & Output Masks

The data set used in this thesis is Oxford-IIIT Pet Dataset [6] which is a data set containing 7349 images of 12 different cat breeds and 25 different dog breeds. It has roughly 200 images of each breed. The goal of the dataset is to extract a mask where every pixel in the output image is classified into one of the three classes: foreground, background, and boundary. Where foreground is the part of the image where the animal is, background is all the pixels that are not part of the animal and boundary is the pixels on the edge of the background and animal. The output image from the network contains three channels, one for each class. Each pixel in each channel contains the probability between 0 and 1 of every pixel belonging to that class.

3.1.1 Thresholding

The output from the neural network shows the probability of the different pixels being part of a certain class based on the intensity of the pixel. To determine which classes the pixel belongs to thresholding has to be applied. Thresholding is the process of determining a threshold value and then setting all the pixels above this value to 255 and all values less than the threshold value to 0. This will turn the image into a mask where all the pixels of a class are white and the rest black. In this thesis, thresholding was automated by using the mean adaptive thresholding.

3.1.2 Ground Truth

There are different possible golden data to use as a baseline to compare the results with: the actual labeled input images, the output images from the perfect non-quantized network, or the 8-bit quantized network. This thesis mostly focuses on the 8-bit quantized output but also discusses the relationship with the labeled input images.

3.1.3 Output Masks

The output from the neural network is a tensor with a channel dimension of three. These three different channels represent different features of the network. The first channel predicts the location of the body of the animal also called the foreground. The second channel predicts the background and the third channel predicts the boundary between the first two predictions. The final prediction masks were extracted from the output in two different ways providing two different interpretations for the masks.

The first one gives an output image where each pixel is assigned to the class it has the highest probability to be. This only lets each pixel belong to one class and the classes cannot overlap. To get the final predicting mask, the function *argmax* is used on the three output channels to get one output channel.

$$\arg\max_{x \in S} f(x) = \{x \in S : f(s) \leq f(x) \text{ for all } s \in S\}$$

(a) Input image of the animal

(b) Prediction mask of the animal

Figure 3.1: Example when using *argmax* to get one mask for the classes combined

The other is to extract one image for each of the classes and do thresholding on it to get the mask. To perform the thresholding automatically and well for different images mean adaptive thresholding [12] was used. This gives a more general case where each pixel can be part of two classes.

Figure 3.2: Example when using thresholding to get one mask for each class in their own image

3.2 Error Metrics

To be able to evaluate the result an error metric had to be chosen. This thesis uses several methods since they all have different advantages and disadvantages.

3.2.1 Mean Average Error

The first of the error metrics is mean average error (MAE) which just calculates the average error for all of the pixels. The mean average error is used on the output image before it is masked and calculated using:

$$MAE = \frac{1}{n}\sum |Y_i - \hat{Y}_i|$$

Where Y_i is a pixel in the golden output image, $\hat{Y}_i$ is a pixel in the output image from the network and n is the number of pixels that was used for the calculation. MAE gives a general idea of how much error the output image has and outliers will not affect the MAE significantly. This is not always an advantage since the outliers are likelier to affect the mask.

3.2.2 Mean Squared Error

Another error metric used is Mean Squared Error (MSE) this is as MAE calculated on the unmasked output image. This squared the error for each pixel so that bigger errors affect the error more. It is calculated using:

$$MSE = \frac{1}{n}\sum (Y_i - \hat{Y}_i)^2$$

Where Y_i is a pixel in the golden output image and $\hat{Y}_i$ is a pixel in the output image from the net. It fits this data because in this case, small errors won't affect the output notably while large pixel errors are a lot more likely to change what each pixel is classified as.

3.2.3 Jaccard Index

The Jaccard index was also used, in this thesis referred to as accuracy where the mask or masks were extracted. Then the accuracy of the new mask or masks was calculated against the golden mask or masks for each class using:

$$\frac{new\ mask \cap golden\ mask}{new\ mask \cup golden\ mask} = \frac{TP}{TP + FP + FN}$$

Then it adds the accuracies of all the classes together and divides it by the number of classes to get the average accuracy. This divides the pixels correctly classified as part of the mask (TP) by the pixels that should be part of the mask(FN) and the pixels that were wrongly classified as part of the mask(FP). This error metric provides a good way to evaluate the result.

When calculating accuracy for three classes a variation of this where the accuracys were squared before they were added was also tried. To compensate for the cases where the accuracy of one of the classes was significantly lower than the accuracy of the others.

3.2.4 F1-score

F1-score is similar to accuracy and works well in the case where some of the classes contain more pixels than others. It is calculated on the extracted mask or masks using the formula:

$$\frac{TP}{TP + \frac{1}{2}(FP + FN)}$$

The F1-score is also calculated using the average of the different classes.

3.3 Finding the Optimal Solution

Since the network has 21 different layers where both the precision of the weights and the input tensors can be reduced, there are a lot of different combinations of bit optimizations. Some bits affect the result more than others and some layers contribute more to the power consumption than the others. Since there are 42 locations where there can be 8 different values 8^{42} different combinations could be calculated which is unrealistic. Instead, we explored two algorithms that reduce the computational complexity: a tree-like algorithm and a bit remover algorithm. These algorithms probably missed the most optimal solutions. This was considered acceptable because they most likely reduced the power consumption by almost as much as the actual optimal solution while reducing the number of possible solutions significantly.

3.3.1 Tree-like Algorithm

One of the tested algorithms is a tree-like algorithm where a criteria is set. This criterion can be a maximum MSE or MAE or a minimum accuracy or F1-score for the average of all the classes or one of the classes. After that, it gets the perfect solutions i.e. golden data to compare the different solutions to. Then it starts to lower the precision of the last layer to one less and if that passes the criteria it lowers it one more time, and so on. When it fails, it resets the bits in that layer to 8 bits but reduces the precision in the next layer, and if that passes the criteria it starts to loop the last layer again. This goes on until it has gone through all the bits. It tries to lower both the number of bits in the weights and the tensors. The algorithm starts the iteration for the tensors before moving to the weights.

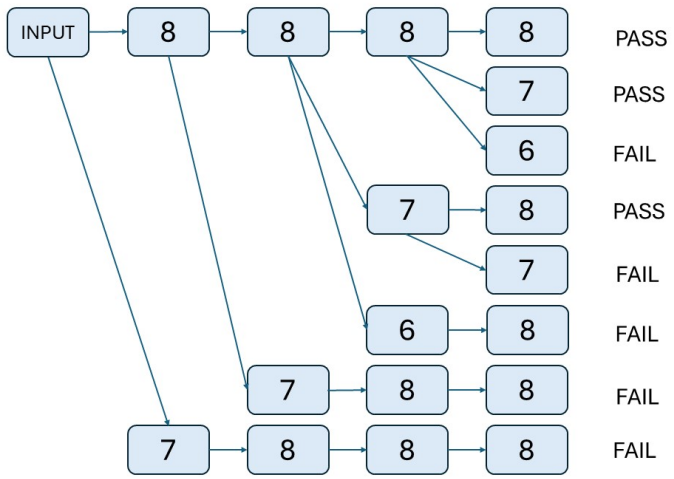


Figure 3.3: Simple illustration of the algorithm for four compute layers

3.3.2 Bit Remover Algorithm

Another algorithm used was a recursive method. In this method, the amount of bits to remove and a starting precision was specified. Then the method tried to reduce the precision of all the weights and input tensors in all of the layers one by one to calculate the criteria for all of the different solutions. Then it continued with the solution that fulfilled the criteria the best and called itself to remove another bit until it had removed as many bits as specified from the beginning. This algorithm was also able to be adjusted to be more likely to reduce the bits that reduced the power consumption more. This was done by dividing or multiplying the criteria by $\log_{10}$ of the number of operations for that layer, allowing it to reduce the power consumption more if a few bits were to be removed. This will probably lose the best solutions since all the bits removed are independent of each other, but it is a fast way to find good solutions. Since it only got 336 possible bits to remove it has a maximum of 12348 different combinations to calculate. Another strength is that it can be biased to remove the bits that reduce the power consumption more if wanted.

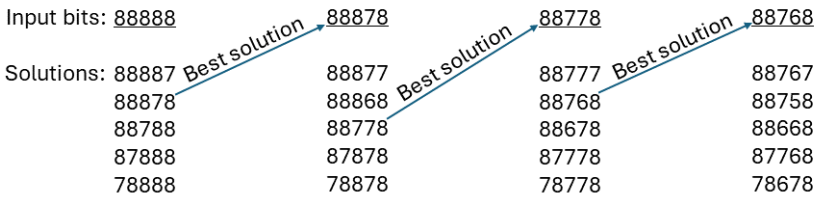


Figure 3.4: Example of the bit remover algorithm with five possible bits to remove; the best solution becomes the input for the next iteration.

Another way to reduce the number of possible solutions is to keep the precision to 8, 4, 2, or 1 bit. This will make the inputs compatible with different kinds of multipliers that usually have the same input sizes. It also keeps the most hardware-friendly solutions but also removes a lot of good solutions.

Every input image has its own optimal solution of bits removed. This means that while one solution might provide good results for that image it may produce much worse results for another image. To counteract this the optimal solution was developed using a set of 30 images of different cats and dogs. The criteria were calculated for all of these and a bit was removed based on the average of the criteria for these images. This creates a more universal solution that will perform well on most images instead of great on one and bad on most of the others. Optimizing based on 30 images was deemed enough partly because of the limitations of the computers that had to run the algorithms. Optimizing on more images might provide a little better results but requires more computing time.

3.4 Energy Estimation

To get a good understanding of how much energy consumption is saved by the quantization of the weights and input tensors for the neural network, an energy estimation for inference of the network needs to be determined. Since the relationship between the energy required for the data transport and the MAC operations is unknown, this can be specified when calling the energy estimation function. Everything inside a MBConv block is assumed to be stored internally in the memory and the data that flows between them is assumed to be stored externally. Table 3.1 describes the energy relationship for different simulations.

Relative cost per bit			
Different Cases	External Memory	Internal Memory	MACs
Case 1	20	1	1
Case 2	40	2	1
Case 3	80	4	1
Case 4	160	8	1

Table 3.1: Energy ratio for different operations

$$\frac{12}{12 + \frac{1}{2}(2+4)} = 0.8$$

After the relationship for energy estimation per operation is determined, the energy can be calculated. The formula works by creating relative energy estimation for each operation type in Table 3.1. This is done for all the 21 stages in the network, where all of the seven MBConvs consist of three stages each. For example, if there are 100 MAC operations and 5 external memory transfers and the relative power usage for the operations is 1:20, they will now both have the relative energy 100. In this example, the bit precision used for the weights is 5 and the precision for the input tensor is 7. The precision for the values that are written to the external memory is 6 and the new energy estimation will be: $100 \times \frac{5}{8} \times \frac{7}{8} + 100 \times \frac{6}{8} = 130$. The original 8-bit precision energy estimation was 200 and the energy saved becomes $1 - \frac{130}{200} = 0.35$.

$$\text{energy saved} = 1 - \frac{\text{new energy estimation}}{\text{original energy estimation}} =$$

$$1 - \frac{\sum_{k=1}^{21} MAC_k \frac{b_{w,k}}{8} \times \frac{b_{i,k}}{8} + \sum_{k=1}^{21} INT_k \frac{b_{i,k}}{8} \times R_{int} + \sum_{k=1}^{21} EXT_k \frac{b_{i,k}}{8} \times R_{ext}}{\sum_{k=1}^{21} MAC_k + \sum_{k=1}^{21} INT_k \times R_{int} + \sum_{k=1}^{21} EXT_k \times R_{ext}}$$

Where MAC_k, INT_k, and EXT_k are the total number of operations of that type in a certain stage k. b_w and b_i are the precision used for the weight tensors and input tensors for a certain stage. The variables R_{int} and R_{ext} are the relative cost per bit for reading or writing to the internal and external memory, which can be seen in Table 3.1 for the different cases.

3.5 Program Design

To perform inference on the convolutional neural network and be able to change bit precision in a certain location in the network, a software program had to be designed and built. To reduce the computation time, the computational heavy functions are written in C++ and the rest in Python. Pybind11 is used to create Python bindings for the code in C++. The program uses the bit remover algorithm from section 3.3.2 to remove bits where the accuracy decreases the least. The inputs to the program are the weights, the images, and setting parameters to determine which type of simulation to execute. This will enable the program to handle different kinds of simulations and therefore a broader range of results. To easily compare the outputs of different simulations, the program has three outputs: plots, images, and raw data. The objective of the program is therefore to use the bit remover algorithm on a set of images to remove bits that affect the accuracy the least for different setting parameters.

3.5.1 Pybind11

Pybind is a framework for combining types in C++ and Python. This enables the network to use both C++ and Python which greatly reduces the time it takes to run inference.

3.5.2 Program Layout

The architecture consists of three building blocks, whereas two are written in Python and one is written in C++. As explained earlier, writing the computationally heavy part in C++ greatly reduces the computation time for this program.

As seen in Figure 3.5, the program takes three inputs. The input images are two sets of images, one to perform on the bit remover algorithm and one to test accuracy on. These images are part of the Oxford-IIIT Pet Dataset [6] which is an open-source library for images on pets. This is also the data which the neural network originally was trained on. In most of the simulations, there were 30 train images and 20 test images. The weights are from the quantized 8-bit model, which originates from the floating-point model but has been quantized to 8 bits using tensorflow. The setting parameters are inputs to determine which kind of simulation to run. For example, the simulation can choose not to quantize below 4 bits or it can remove 2 bits at a time, instead of 1 bit.

The program itself consists of two parts. One part simulates the optimal solution and the other quantizes the weights and input tensors and then runs the neural network together with a C++ file. The simulation for the Optimal Solution Algorithm loops through so many bits the program wants to remove from the network. The network has seven stages of MBConvs, each containing three steps of convolution. As said before, the computational heavy part of the neural network is executed in C++. It includes 1x1 convolution, depthwise convolution, the mul and add operations, and quantization of the weights and input tensors. The algorithm to find the optimal solution is implemented in "Simulation for Optimal

solution" in Figure 3.5.

The output of the program is given in three different formats, plots, data files, and images. The plots provide information about how different metrics are affected by bits removed and the energy saved. Data files enable the data of a simulation to be saved and later reused. These include bit precision for the different weights and tensors, accuracy, F1-score, their respective standard deviation, and estimation of the power consumption. For example, if a new plot is made or if a table of values is constructed. Images give the reader a sense of how the prediction mask looks when different amounts of bits are removed.

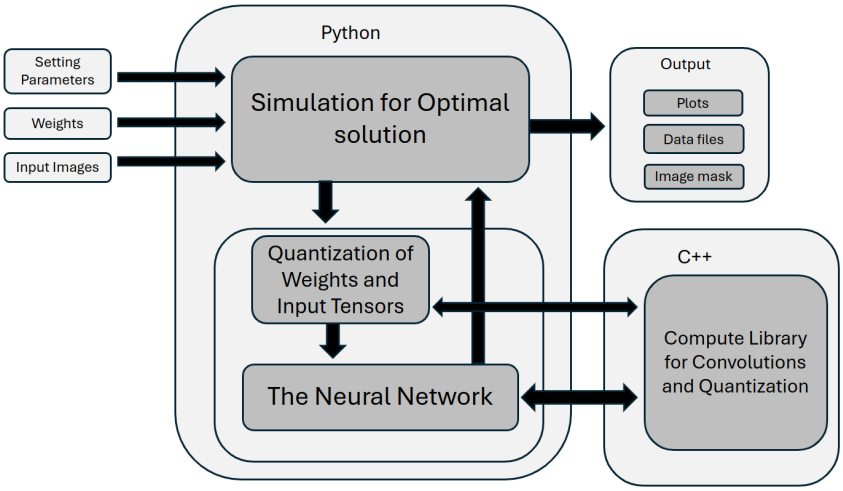


Figure 3.5: Program Design

3.5.3 The Neural Network

The neural network is built in Python using function calls to C++, which is possible with Pybind11. The flowchart in Figure 3.6 describes how the data flows during one MBConv iteration. It receives the input from Python and then quantizes both the weights and the input tensor in C++. Convolution is done after quantization and the output from the convolution goes back into the Python program. The MBConvs used in the U-net differs a bit depending on which layer it's placed in as explained by the different MBConvs in Section 2.3.3. It could be upsampling, concatenate, batch normalization, or just data transfer from one stage to another.

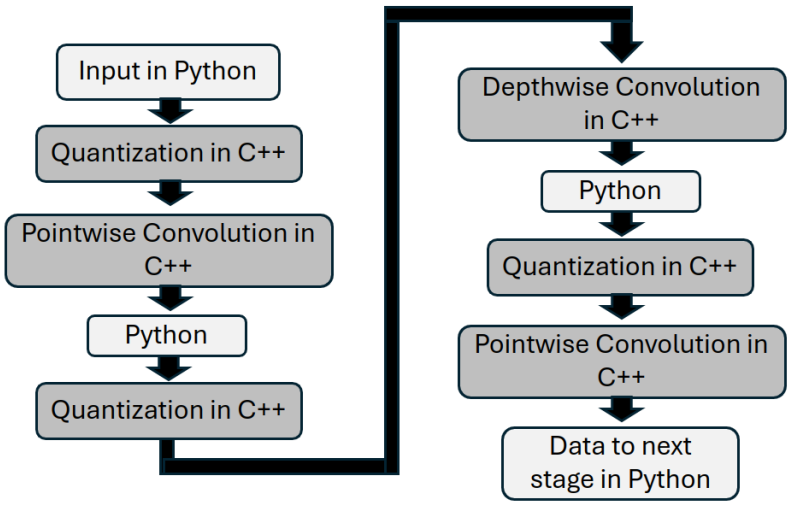


Figure 3.6: Typical flowchart for one MBConv

3.5.4 Quantization

As seen in Figure 3.6, there are two C++ functions to quantize the weight tensors and input tensors. One of them takes three-dimensional arrays and the other takes two-dimensional arrays. Both are needed since the input tensors are three-dimensional and weight tensors for depthwise convolution too, while the weights for pointwise convolutions have two dimensions. Both functions quantize the values in the same way using the algorithms shown below in algorithm **??**:

Algorithm 1 Quantization Algorithm

1: $w \leftarrow (w + 2^{n-1}) \gg n$
2: **if** $w \geq 2^{7-n}$ **then**
3: $\quad w \leftarrow 2^{7-n} - 1$
4: **end if**

Where w is the weight or input tensor and n is the number of bits that shall be removed. This will make sure that all weights are represented as a value of $8 - n$ bits. This means that fewer bits have to be used by the multipliers as well and fewer bits have to be transported from the memory. Both reduce power consumption. The network keeps track of how many bits the weight tensors have been reduced by to compensate for it after the operations by left-shifting the output.

3.6 Energy Estimation for Approximate Multipliers

The energy estimation when using approximate multipliers was calculated using the power consumption given by the EvoApproxLib [11]. Since only the calculations are affected by approximate multipliers and not the data transport the calculation part compared to the total energy consumption was calculated using:

$$C = \frac{O_n - O_1}{O_n + R_i + R_e \times n + W_i + W_e \times n}$$

Where C is the fraction of the energy that is affected by changing the multipliers, O_n is the number of total operations in the network, O_1 is the number of operations in the stages using normal multipliers instead of approximate, in this case, the first stage. R_i is the number of internal reads, R_e is the number of external reads, W_i is the number of internal writes, W_e is the number of external writes and n is the ratio of how much energy it costs to interact with an external memory instead of the internal. The energy for the different approximate multipliers is then calculated using:

$$\frac{P_{approx}}{P_{original}} \times C + (1 - C)$$

Where P_{approx} is the power consumption for the approximate multipliers according to the EvoApproxLib. $P_{original}$ is the power consumption for an accurate multiplier also given by the EvoApproxLib. This only reduces the part of the energy consumption from the multipliers: C, and not he part of the energy consumption from the data transport: $(1 - C)$.

Chapter 4
Results and discussion

In this section, we present the results and how they should be interpreted. Some of the results based on metrics like F1-score or MSE might be difficult to interpret. Therefore, images of the prediction mask is used to display the results as well. The section also includes a discussion of all of the results to be able to weigh the advantages and disadvantages of the different methods and error metrics.

4.1 Choosing the Algorithm

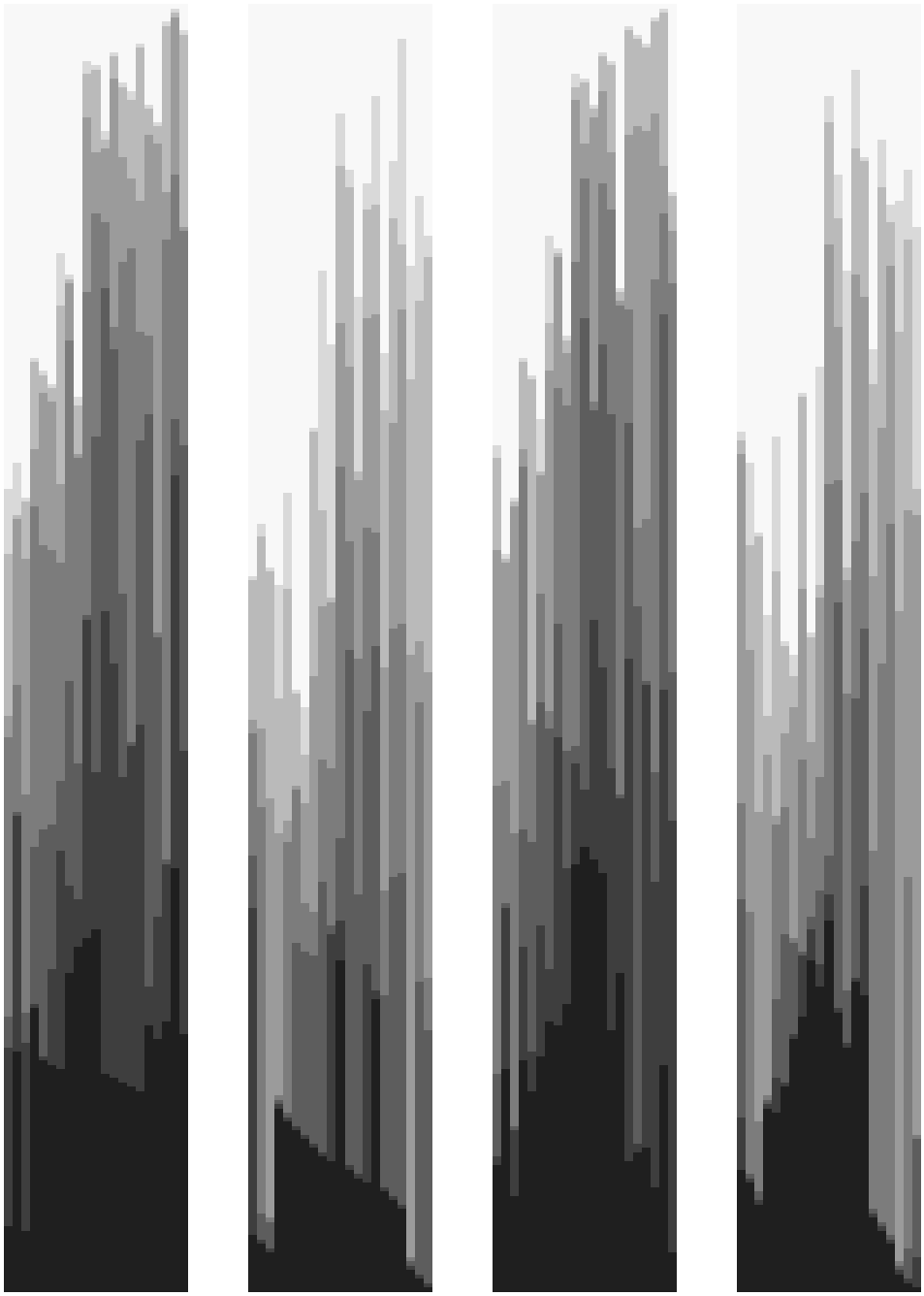
After trying both of the algorithms for finding an optimal solution in bit precision; the bit remover algorithm was deemed superior to the tree-like algorithm. Even if the tree-like algorithm theoretically was able to find better solutions. It had too many solutions to test to be able to remove enough bits to get a result that made a significant impact on the power consumption. When using the bit remover algorithm, it had to calculate one inference to obtain the golden data and then 42 different inferences per bit removed. Since each inference took about 0.2 seconds this was a quick way to remove bits with the positive that the time it took was linear.

The runtime for the tree-like algorithm is quicker in the beginning because to remove one bit, it only has to calculate inference for a single solution. The problem is that the runtime increases exponentially when lowering the criteria. For example, if 7-bit precision for all of the weights and tensors is the best solution it has to try 8 and 7-bit precision for 42 different variables. Meaning that it has to calculate over 4 trillion different networks. If this was a solution for the bit remover algorithm it only would have to calculate 1764 different networks. This does not necessarily mean that the bit remover algorithm will be faster in every case but in the cases where the power consumption will be lowered considerably, it will. Even when adding more constraints like limiting the number of bits in the precision to 8, 6, 4, 2, or 1 it takes too long time to be able to find a good solution.

When using the bit remover algorithm we decided against limiting it to 8, 6, 4, 2, or 1 bit precision because the runtime was deemed low enough. This will most likely let it find a solution that gives a lower power consumption while not affecting the output more than if it were unconstrained. The low runtime also allows the

algorithm to optimize based on 30 different images to produce a more universal solution. When optimizing based on different input images, solutions tend to reduce precision in the same stages. Most of them have different optimal solutions when the algorithm has removed enough bits to reduce the power consumption significantly.

4.2 Histogram over Removed Bits



(a) **(b)** **(c)** **(d)**

Figure 4.1: Representation of precision of the weights and input tensors in each stage

Figure 4.1 illustrates where the algorithm removes bits in both the weights and tensors when optimizing for a high accuracy and a low MSE. Each column represents a different stage in the network, the first 21 represent the weights for the different stages (Figure4.1a and Figure 4.1c) and the last 21 represent the input tensors (Figure 4.1b and Figure 4.1d). Each row in the histogram represents an iteration of the bit remover algorithm. The intensity of each pixel represents the precision, where a brighter pixel uses more bits for the weight or input tensor. Figure 4.1a shows which bits are removed in the weights and Figure 4.1b shows the bits removed in the tensors when optimizing for a high accuracy. Figure 4.1c shows the bits removed in the weights and Figure 4.1d shows the bits removed in the tensors when optimizing for a low MSE. When two different bits removed get the same accuracy the algorithm removes a bit in the precision of the weights before a bit in the precision of the input tensors. If both are weights or input tensors it removes the bit in the earliest of the stages, this explains the unusual behavior when most of the bits have been removed. This might not be optimal but as concluded before the network won't provide any usable results when reaching this level of quantization so it does not matter.

Figure 4.1 clearly shows that the bit remover algorithm is a lot more likely to remove precision from the weights than from the tensors in both cases. Removing precision from the tensors lowers the power consumption more than reducing the precision of the weights. This is because there are fewer weights than tensors that have to be read and only the tensors have to be saved back to the memory. We have also tried to optimize a little by hand to try to remove precision on the tensors earlier. Doing that affects the output image significantly more than if we were to reduce the precision of the bits for the weights.

4.3 Pre-masking: Comparing MAE and MSE

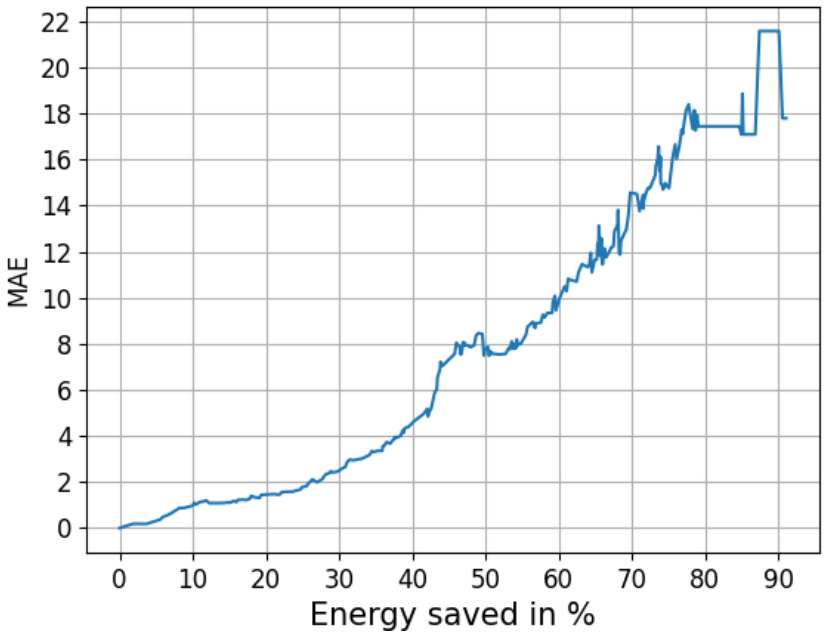


Figure 4.2: Plot showing mean absolute error plotted against saved energy

Figure 4.2 shows the mean absolute error, explained in Section 3.2.1, for each pixel calculated on the output from 20 different images that the bit precision was not optimized for. The weird behavior starting at 85% saved consumption is due to the results being so bad that the output image is almost unusable.

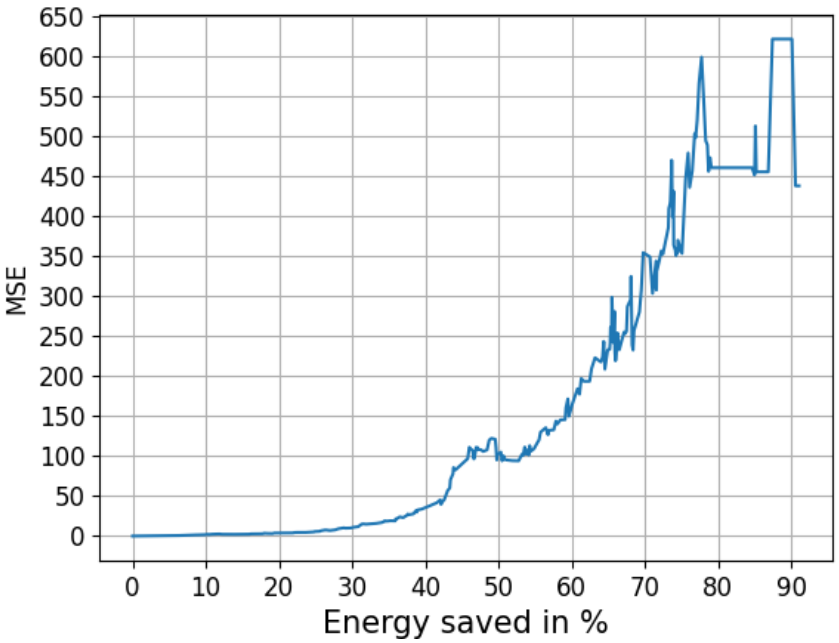
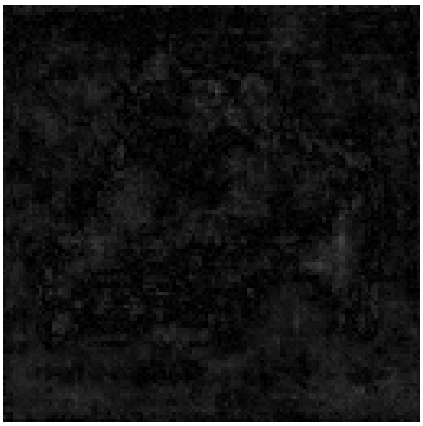


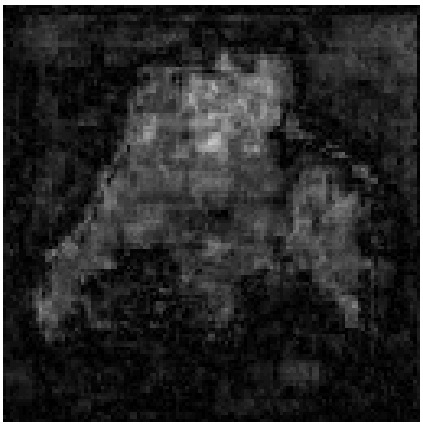
Figure 4.3: Plot showing mean squared error plotted against saved power

Figure 4.3 shows the mean squared error for the same 20 images as the mean absolute error is calculated. It also uses the same bit precision for the weights and tensors. As expected the MSE is a lot higher than the MAE. The mean absolute error stagnates at around 17 while the mean squared error stagnates at around 460. If the error was uniformly distributed over all the pixels in the output image MSE would be $17 \times 17 = 289$, which means that the error varies a lot. We can see this in Figure 4.4 where the pixel error seems to be higher in the pixels near pixels classified as another class. This makes sense since all the class's probabilities usually are higher when the pixel is closer to pixels belonging to other classes.

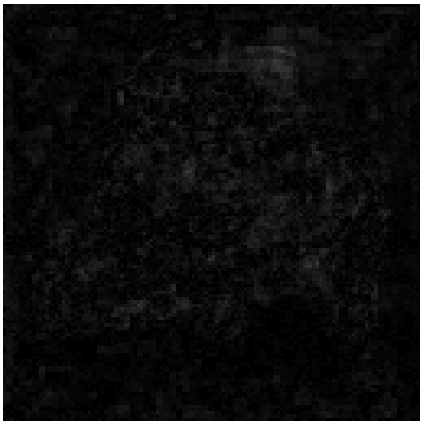
Since mean square error not only shows how much error the output image has but also shows if some pixels have a more extreme error than others, it was deemed superior when compared to mean absolute error. A pixel with a high error is much more likely to impact the class that it is classified as.



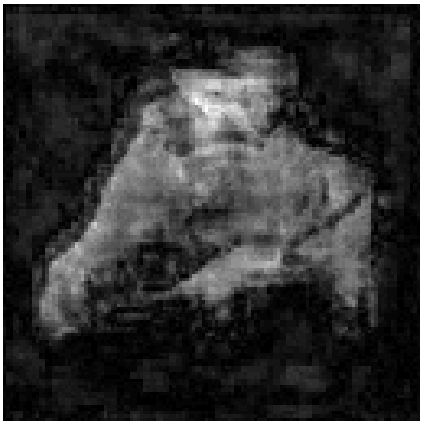
(a) Error map for the foreground with 60 bits removed



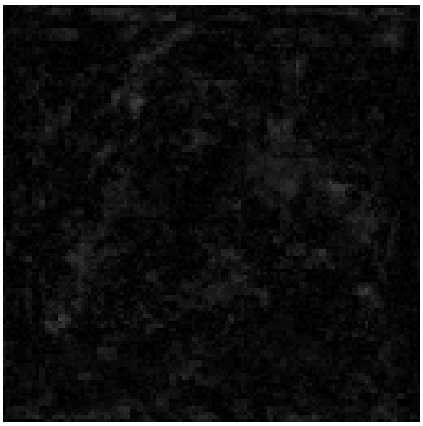
(b) Error map for the foreground with 120 bits removed



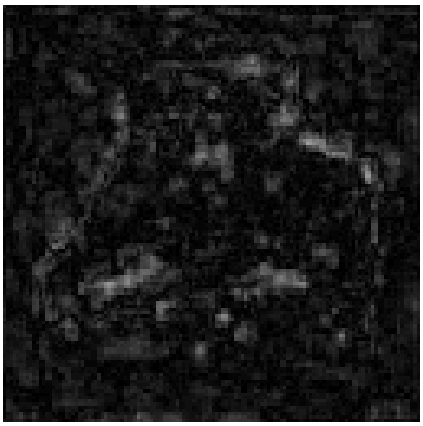
(c) Error map for background with 60 bits removed



(d) Error map for background with 120 bits removed



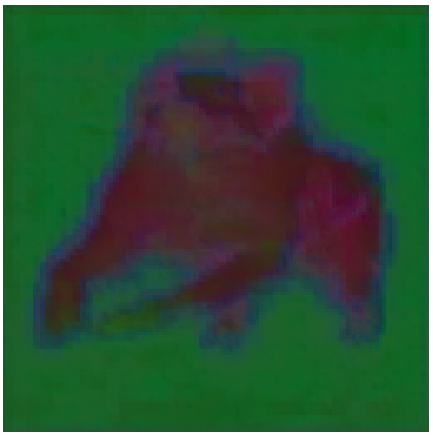
(e) Error map for the boundary with 60 bits removed



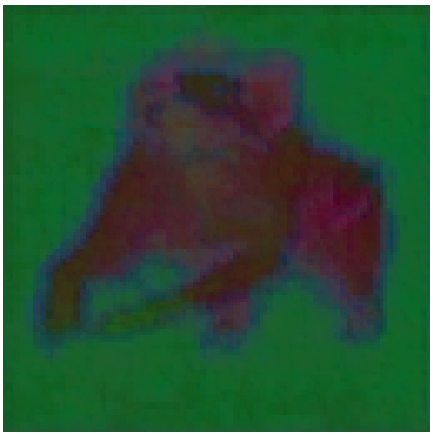
(f) Error map for the boundary with 120 bits removed

Figure 4.4: Error maps for the three classes for 60 and 120 bits removed

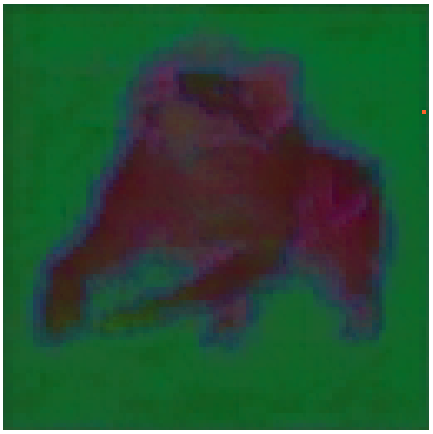
In Figure 4.4 the absolute error for the three different classes can be seen. The higher the intensity of the pixel the higher the error for the pixel. To make them easier to see; all the error maps are normalized towards the maximum error for 120 bits removed in the background class. Since it was the class with the maximum absolute error in a single pixel. This shows that the errors are not evenly spread out over the output image but tend to concentrate toward the foreground for both the foreground class and the background class. For the boundary class, the errors seem to be the highest at the boundary. This probably occurs because the probability of the pixels not belonging to the boundary class is low unless they are near the pixels that are supposed to be classified as the boundary class. Figure 4.4 also helps to show that the error increases exponentially when removing precision, for example, Figure 4.4d is significantly brighter than Figure 4.4c. Both Figure 4.4b and 4.4d have a clear concentration of errors in the pixels belonging to the foreground class which also can be seen in Figure 4.5. When the foreground starts to become greener and less red meaning that the network is less certain of the pixels class when removing precision.



(a) Output image with 0 bits removed

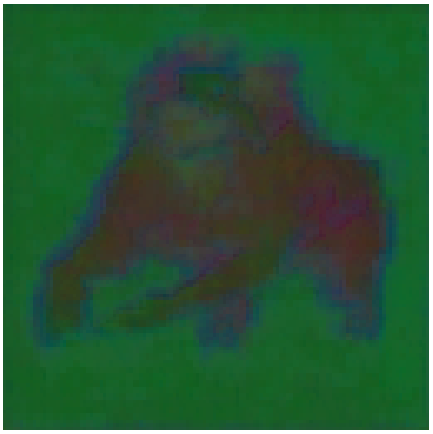


(b) Output image with 30 bits removed

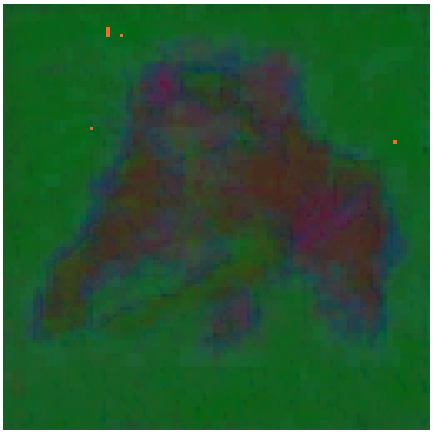


(c) Output image with 60 bits removed

(d) Output image with 90 bits removed



(e) Output image with 120 bits removed



(f) Output image with 150 bits removed

Figure 4.5: Output images for different levels of precision scaling

Figure 4.5 shows the output image when removing a certain number of bits from the precision, the bits are removed using the bit remover algorithm based on the accuracy. Since there are three classes illustrating each class as a different RGB channel works perfectly. In this case, the foreground class is represented by the level of red in each pixel. The background class is represented by the level of green in each pixel and the boundary class is represented by the level of blue in each pixel. This gives a good idea of what class most of the different pixels will be classified as. The network is trained to not let any pixels get a value under 0 but since the values of the weights and input tensors are changed pixels can get a value below 0. This is the case in Figure 4.5c and especially Figure 4.5f where some pixels are bright red. Figure 4.5 clearly shows the degradation of the network's certainty of which class a pixel belongs to.

4.4 Post-masking: Comparing Accuracy and F1-score

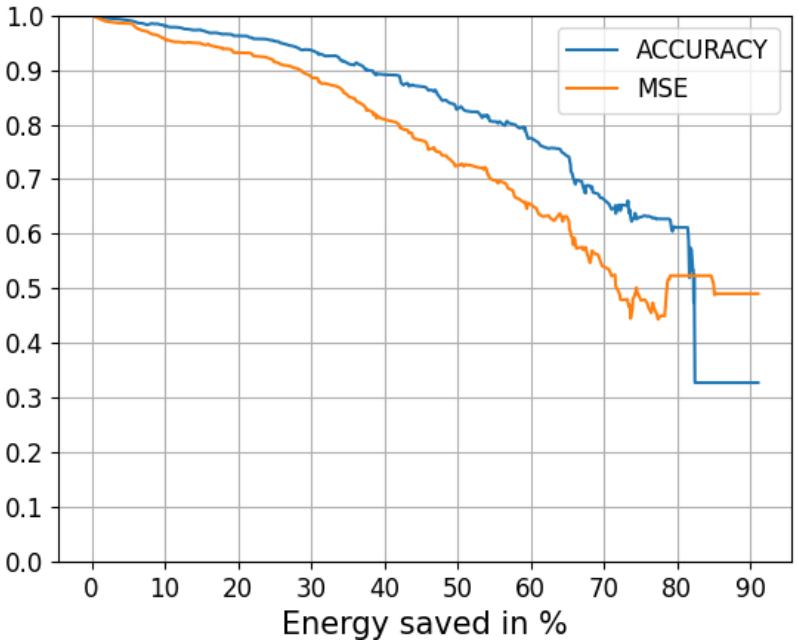


Figure 4.6: Accuracy and F1-score plotted against the power consumption

Figure 4.6 shows the accuracy and F1-score using the same precision and test images plotted against the power consumption. Both of these are calculated using very similar formulas and as expected both lines are relative to each other with the accuracy line being a bit lower than the F1-score line. They are hard to compare

since they say about the same thing but we have decided that the F1-score is the better error metric. Because it is affected more by the pixels classified correctly than accuracy and because it is a more common error metric when evaluating machine learning tasks.

4.5 Optimize for low MSE or high Accuracy?

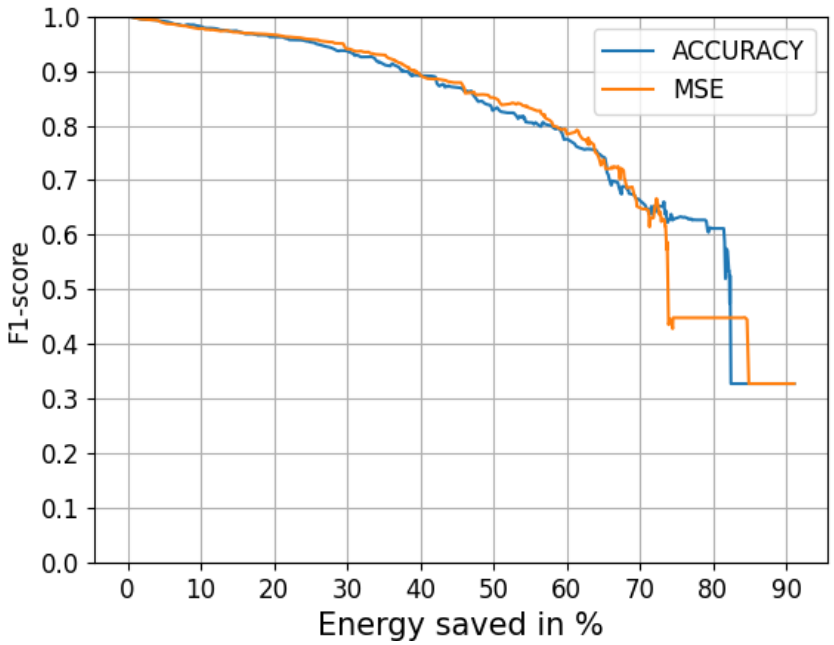


Figure 4.7: F1-score for different bit optimizations

Figure 4.7 shows the F1-scores of the image when trying to optimize on 30 images for a high accuracy and a low MSE. The outputs were then tested on 20 different images. This test was done to compare different criteria to optimize the bits. In Figure 4.7 the F1-scores are plotted against the power consumption since the goal of the algorithm is to lower the power consumption as much as possible. When reaching a really low power consumption the F1-score drops significantly but as before when this happens the results are probably unusable anyway. According to the graph sorting for a low MSE is sometimes better than optimizing for a high accuracy but not always and even when it got a better F1-score the score is not that significantly higher.

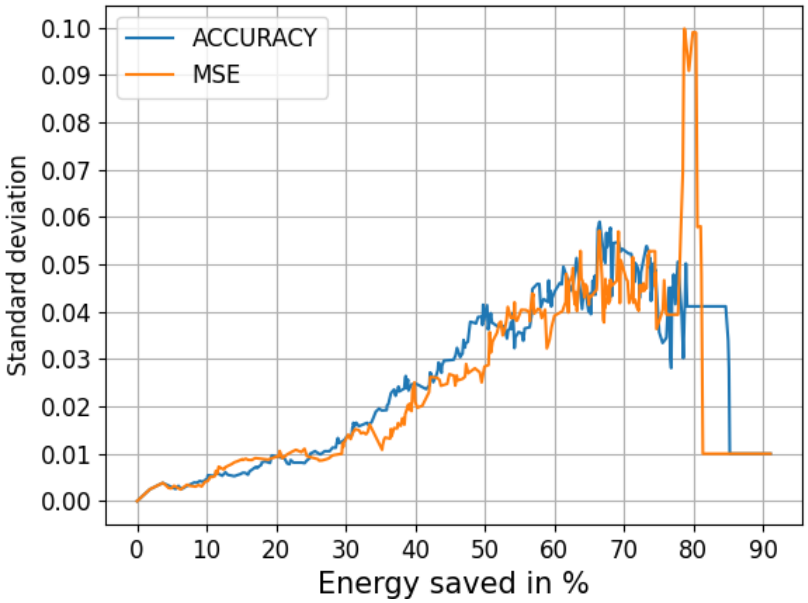
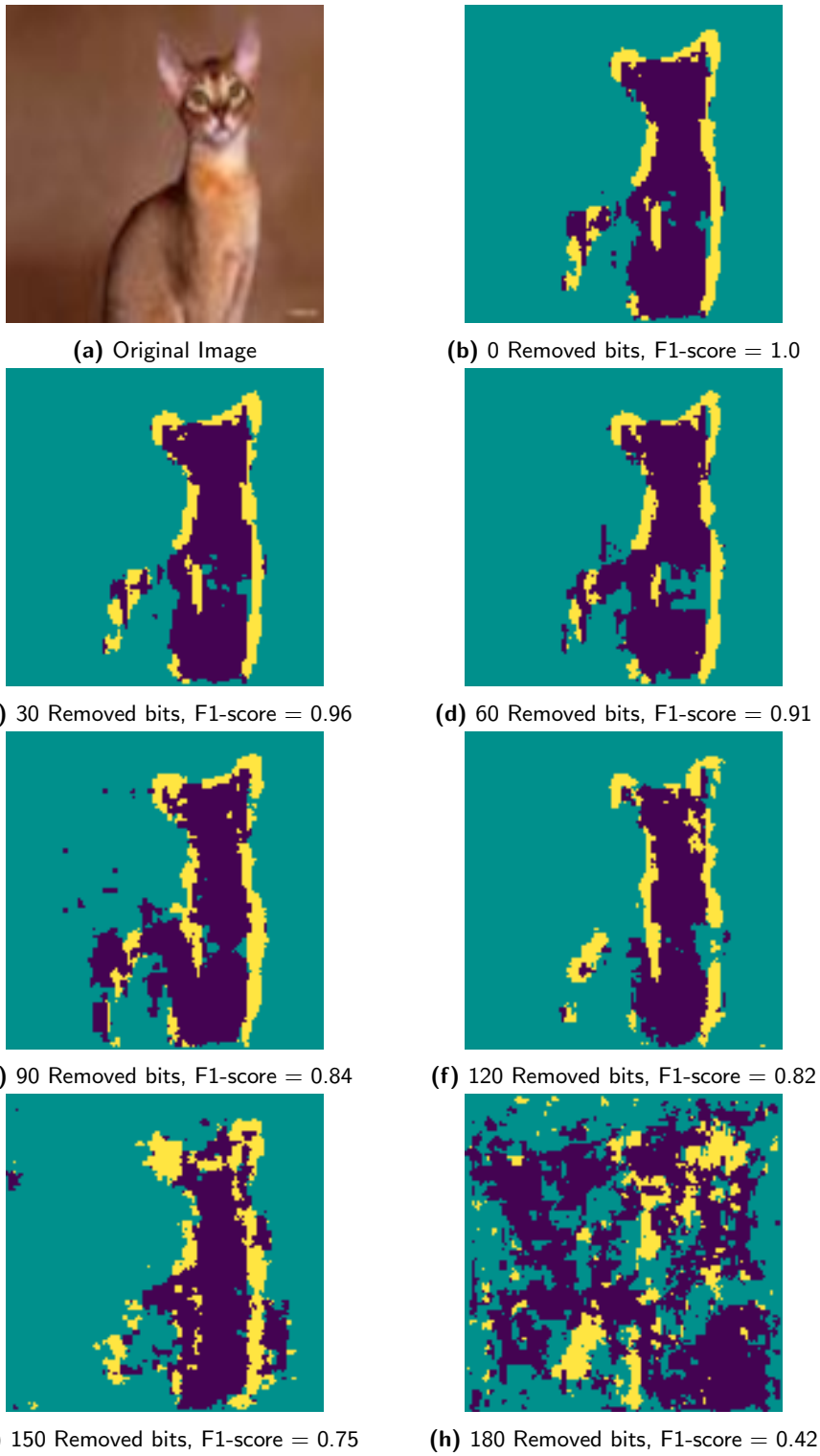


Figure 4.8: Standard deviations for different optimization criteria

Figure 4.8 shows the standard deviations for the F1-score when optimizing and testing on the same images as before. It shows the standard deviation both when optimizing for high accuracy and low MSE. It shows about the same as in Figure 4.7 that optimizing for a low MSE most of the time is marginally better than optimizing for high accuracy. Therefore we can't confidently conclude which optimization criteria that is better. Optimizing for a lower MSE probably works better when removing more bits since it optimizes for a solution that affects the output image as little as possible. Optimizing for a high accuracy might work better in the beginning since it does not care if the output image is affected badly as long as it provides a high accuracy. With the drawback that it might prove worse in the long run.

4.6 Prediction Mask from Quantized Network

Figure 4.9 shows different outputs as well as the total number of bits removed from the weights and tensors precision from the network for a few different cases. This result gives a hint of how high the F1-score needs to be for an acceptable output. As seen in Figure 4.9h, the prediction mask is worthless and no cat can be seen. The other six prediction masks still give an acceptable prediction mask and depending on what F1-score the application needs, even Figure 4.9f might be unacceptable. This introduces the importance of standard deviation and why it should be used together with the average to create a one-tailed test. The images in Figure 4.9 give an understanding of how the reader should interpret different values of the F1-score in the report.



(a) Original Image

(b) 0 Removed bits, F1-score = 1.0

(c) 30 Removed bits, F1-score = 0.96

(d) 60 Removed bits, F1-score = 0.91

(e) 90 Removed bits, F1-score = 0.84

(f) 120 Removed bits, F1-score = 0.82

(g) 150 Removed bits, F1-score = 0.75

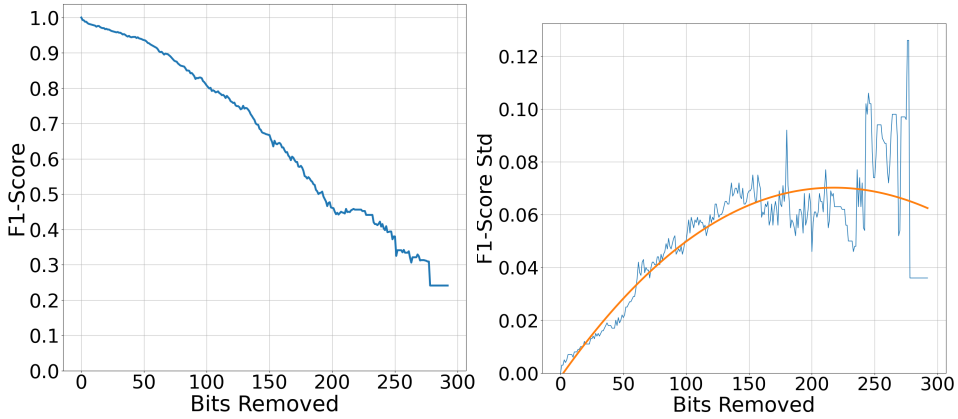
(h) 180 Removed bits, F1-score = 0.42

Figure 4.9: An animal with its prediction mask

4.7 F1-score Plots

Figure 4.10a and 4.10b show how the F1-score and its standard deviation depend on how many bits of precision that has been removed from the network. As explained earlier in section 3.3, the method can remove bits in 42 locations up to a total of 294 of the 336 bits. Half of those are bits for the precision for weight tensors and the other half are precision for the input tensors.

It can be seen that the standard deviation increases when the F1-score decreases until almost 150 bits are removed. The standard deviation then starts to converge while the F1-score keeps decreasing as well. Figure 4.10b shows a volatile standard deviation, which could describe a possible fall after 150 bits and then again would start to increase after 170 bits. Another reason could be that the F1-score starts to get so bad that almost all output prediction masks result in a low score. This would decrease the F1-score but start to increase the standard deviation slowly.



(a) Average F1-score plotted against bits removed

(b) Standard deviation for F1-score plotted against bits removed

The drawback of plotting the F1-score and standard deviation against bits removed from the network is that it does not relate the performance decrease to how much the power consumption has been reduced. Removing bits from different locations in the network affects energy consumption differently. For example, 4 bits are removed from the stage with most MAC operations will save more energy than removing bits from the stage with least MAC operations. Because of the goal to reduce as much energy as possible, comparing the F1-score against energy saved is therefore more appropriate for this thesis.

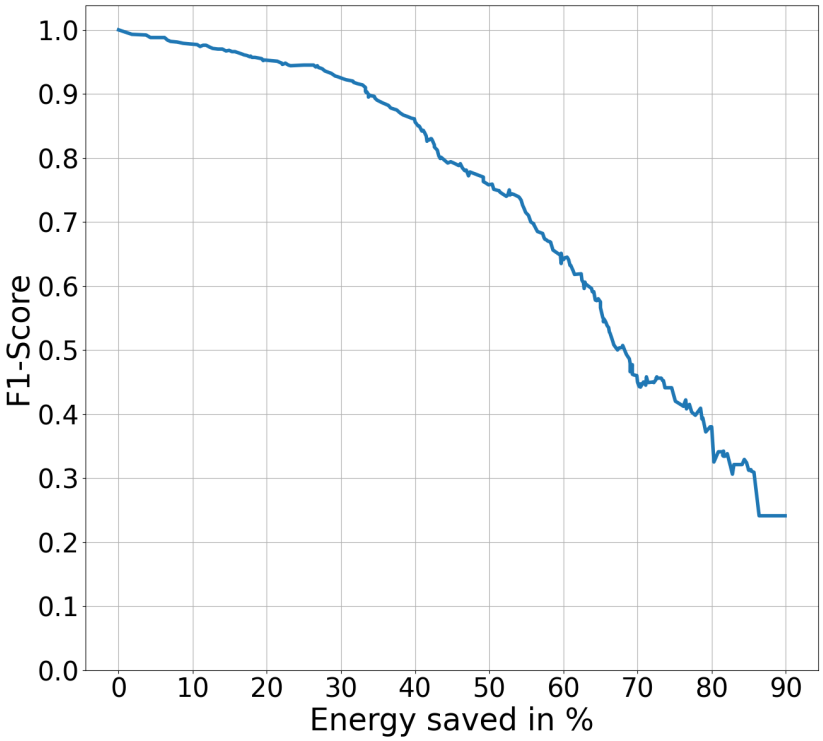


Figure 4.11: Average F1-score plotted against energy saved in %

4.8 Different Energy Estimations

Table 3.1 shows the energy ratio between different operations in simulations and the result is shown in Figure 4.12. As expected the energy saved differs between the cases but not significantly. This can be better understood if looking back at table 2.4. The ratio between the three different operations is 65:14:1. These ratios combined with the energy estimation ratio for the blue line in Figure 4.12 becomes 65:14:20. Now the type of operation that consumes the least energy is interacting with the internal memory. This is also true for the yellow and green lines. For the green line, external energy surpasses MAC and the relative energy consumption becomes 64:56:80. For the red line, MAC operations affect energy the least, with the ratio now being at 64:112:160. This means the external memory will affect energy consumption the most and MAC affect the least.

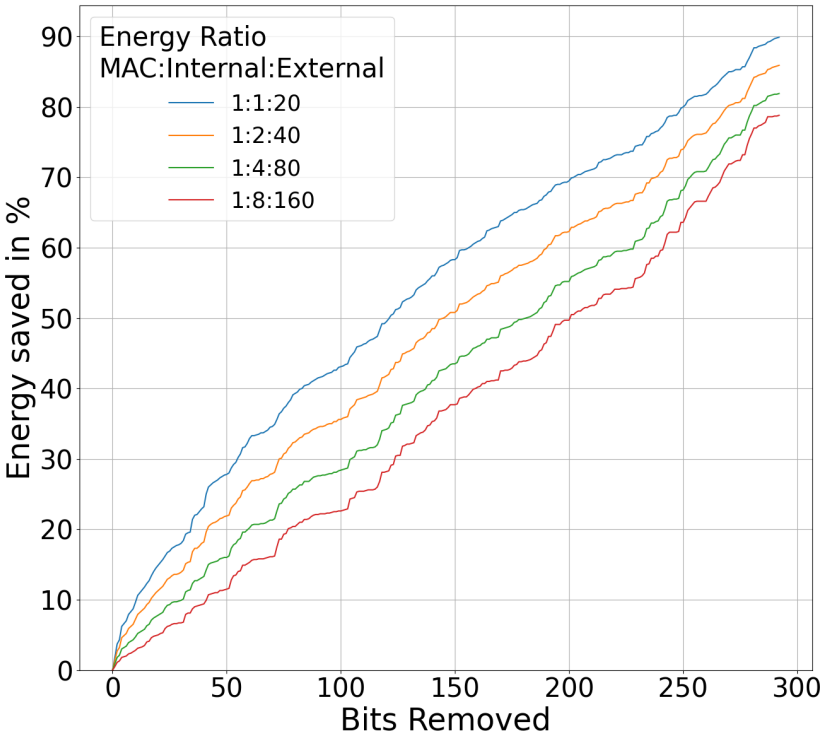


Figure 4.12: Energy saved for different estimations plotted against bits removed

4.9 Different Simulation Parameters

In Figure 4.13a and 4.13b different methods of how to remove bits were tested. The original methods work the same as the original bit remover algorithm explained in section 3.3. It removes one bit at each iteration and goes down to a minimum of 1 bit at each of the 42 locations.

By looking at both Figure 4.13a and 4.13b, the four methods perform about as well when it comes to F1 scores. After 125 bits had been removed, the methods that were prohibited to remove more than 4 bits decreased in performance. Figure 4.13b shows that the method of the even bit, which performed as well as the original method, also performed worse than the original when enough bits were removed. The F1-score needed for different applications differ and if an average F1-score of 0.8-0.75 is enough, the original method performs just slightly better than the rest.

4.10 One-Tailed Test

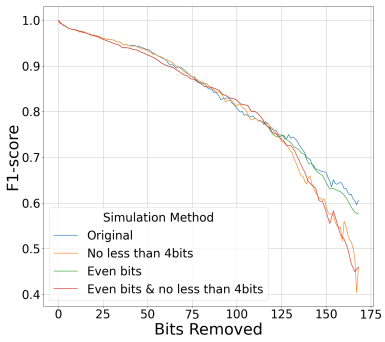
A good average F1-score doesn't need to be a good result. For example, the result needs to be sure that all images perform above a certain F1-score. Including standard deviation in the plot can give more explicit results on what F1-score is needed to be sure that almost all images give acceptable output.

Figure 4.13c shows the average F1-score for its first and second standard deviation. For the first standard deviation, there is an 84% chance that the output F1-score of an image is above the average F1-score subtracted by one standard deviation. For example, if the average F1-score is 0.8 and the standard deviation is 0.05, there's an 84% chance that the F1-score for an individual image is above 0.75. The same applies to the second standard deviation, where in this case only 1 in 50 images have a lower F1-score than 0.70. The standard deviation asserts a certain confidence in the results, meaning that it can guarantee that 98% of all images will be above the lowest acceptable F1-score.

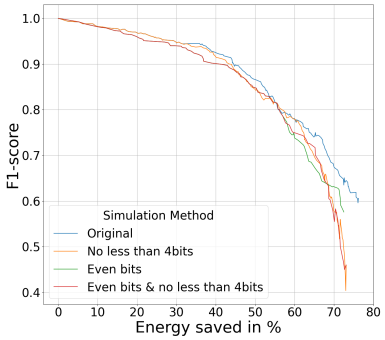
4.11 Approximate Multipliers

Multiplier	F1-score, one mask	F1-score, three masks	Saved energy[%]
1KV9	0.820	0.931	2.52
1KVA	0.545	0.873	5.74
1KVQ	0.432	0.841	12.42
1KX5	0.238	0.724	22.85
1KXF	0.240	0.642	31.58
1L12	0.286	0.471	62.66

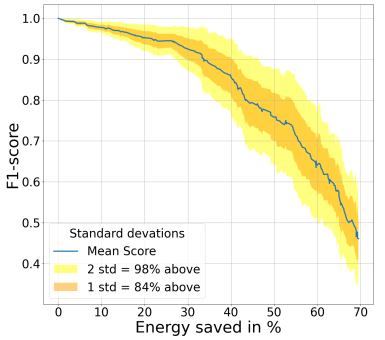
Table 4.1: The F1-score and saved energy when replacing all but the first stages multipliers with approximate multipliers



(a) F1-Score for four different simulation methods plotted against bits removed



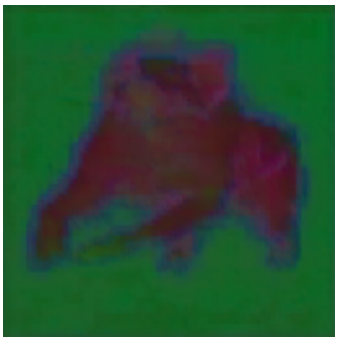
(b) F1-Score for four different simulation methods plotted against energy saved in %



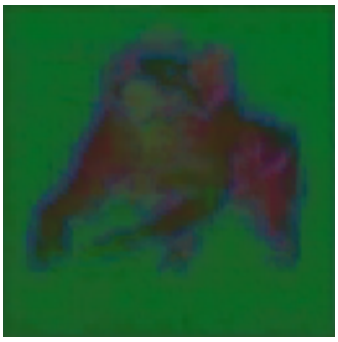
(c) F1-score with Std for plotted against Energy saved in % Simulation Method: Original

Figure 4.13

Table 4.1 shows the results from replacing the multipliers for all except the first stage in the network with approximate multipliers. All the results are tested on the same 20 images as the others and all the weights and input tensors have full 8-bit precision. The saved energy for all the multipliers is calculated using the assumption that the relation between interacting with the external memory requires 20 times more energy than interacting with the internal. The table shows the F1-score both when extracting a single mask and when extracting three different masks. The F1-score when extracting three masks is significantly better than it is when extracting a single mask.



(a) Golden output image



(b) 1KVA output image

Figure 4.14: The output image when using the multiplier 1KVA

(a) Golden single mask

(b) 1KVA single mask

Figure 4.15: The single mask when using the multiplier 1KVA

Figure 4.15 shows the single mask that is extracted from the output images in Figure 4.14. This explains the bad results in Table 4.1 where the F1-score for the single mask becomes low when approximate multipliers are introduced. Since the green representing the background class becomes much more apparent in the middle of the foreground those pixels get classified as background instead of foreground.

This is not the case when extracting a different mask for each of the different classes, this case provides good results. Probably since all the classes still are distinguishable in Figure 4.14b. Figure 4.16 shows all the classes without approximate multipliers in the left column and with the 1KVA multiplier in the right column. Here is apparent that the F1-score is going to be a lot higher than when extracting a single mask.

(a) Golden foreground mask

(b) 1KVA foreground mask

(c) Golden background mask

(d) 1KVA background mask

(e) Golden boundary mask

(f) 1KVA boundary mask

Figure 4.16: The masks for the three classes when using the multiplier 1KVA

Chapter 5
Conclusion and Future work

Here we present our conclusion and some similar interesting areas that can use the results from this thesis to get started.

5.1 Conclusion

The results show that reducing the power consumption by reducing the precision of the weights and input tensors is possible and provides good results. The amount of energy saved may vary depending on the hardware that runs the U-net. For example, the energy estimation varies between different hardware implementations. The results showed a correlation between the power consumption and both the F1-score and MSE. Regardless if the optimizations were done towards a high F1-score or a low MSE the energy savings were similar. However, depending on whether the optimizations were done towards a high F1-score or low MSE the solutions might differ. This shows that different metrics can be used to optimize the U-net.

Initial experiments using approximate multipliers gave inconclusive results. We only evaluated a few different multipliers to test our simulation environment. No optimizations were performed to select the correct multipliers. Much more work is needed to prove if approximate multipliers are useful.

It is difficult to draw an absolute conclusion since the acceptable F1-score depends on the application. As the energy savings introduce a penalty on the F1-score the level of optimization varies with the application. In some cases, a F1-score of 0.8 is enough while in other applications it may be unacceptable. We started with a network that was too small for the task of classifying all pixels as the correct class. This means that even the results when using 8-bit precision are not close to perfect.

5.2 Future work

Our work with approximate multipliers can only be seen as an introduction. There is a lot more to investigate in this area. For example: how putting approximate multipliers in only some stages or how using different approximate multipliers in different stages for the same network affects the network. As well as combining precision scaling and approximate multipliers in the same network to possibly produce even better results than using only one of them.

Reducing the precision in the weights and tensors reduces the power consumption but decreases the performance of the network. This thesis has used a fixed seven-stage network but what if the architecture of the network and the precision were scalable? For example, the network might perform better with a higher precision and fewer stages or a lower precision and more stages.

One method that can be used to increase performance in neural networks after they have been quantized is called retraining which according to [2] works well. Implementing retraining in a network with different bit precision in different layers is difficult and time-consuming. If time permits, this would be interesting to see in future work.

Bibliography

[1] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," *CoRR*, vol. abs/1801.04381, 2018. arXiv: 1801.04381. [Online]. Available: http://arxiv.org/abs/1801.04381.

[2] G. Armeniakos, G. Zervakis, D. Soudris, and J. Henkel, *Hardware approximate techniques for deep neural network accelerators: A survey*, Mar. 2022. [Online]. Available: https://arxiv.org/abs/2203.08737.

[3] *Exploring resnet50: An in-depth look at the model architecture and code implementation*. [Online]. Available: https://medium.com/@nitishkundu1993/exploring-resnet50-an-in-depth-look-at-the-model-architecture-and-code-implementation-d8d8fa67e46f.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[5] C.-F. Wang, *A basic introduction to separable convolutions*, Aug. 2018. [Online]. Available: https://towardsdatascience.com/a-basic-introduction-to-separable-convolutions-b99ec3102728.

[6] O. M. Parkhi, A. Vedaldi, A. Zisserman, and C. V. Jawahar, *Cats and dogs*, 2012.

[7] N. Markuš, *Fusing batch normalization and convolution in runtime*. [Online]. Available: https://nenadmarkus.com/p/fusing-batchnorm-and-conv/.

[8] A. M. Rahmani, P. Liljeberg, A. Hemani, A. Jantsch, and H. Tenhunen, *The Dark Side of Silicon - Energy Efficient Computing in the Dark Silicon Era*. Jan. 2016, ISBN: 978-3-319-31594-2. DOI: 10.1007/978-3-319-31596-6.

[9] *Introduction to weight quantization*. [Online]. Available: https://towardsdatascience.com/introduction-to-weight-quantization-2494701b9c0c.

[10] A. K. Babak Rokh Ali Azarpeyvand, *A comprehensive survey on model quantization for deep neural networks in image classification*, Oct. 2023. [Online]. Available: https://arxiv.org/abs/2205.07877.

[11] E. hardware group, *Evoapproxliblite*. [Online]. Available: https://ehw.fit.vutbr.cz/evoapproxlib/?folder=multiplers%2F8x8_signed%2Fpareto_pwr_mse.

[12] P. D. Team, *Mean adaptive threshold*. [Online]. Available: https://plantcv.readthedocs.io/en/latest/mean_threshold/.