Technion – Israel Institute of Technology

Electrical Engineering Department

Laboratory of Control, Robotics and Machine Learning

Optimizing Prediction Pattern of Zero-Valued Activations in Convolutional Neural Networks

Winter Semester of 2019

Presenters: Ido Imanuel and Inna Batenkov

Supervisor: Gil Shomron

Table of Contents

1. Abstract

Convolutional neural networks (CNNs) compute their output using weighted-sums of adjacent input elements. This method enables CNNs to achieve state-of-the-art results in a wide range of applications such as computer vision and speech recognition. However, it also comes with the cost of **high computational intensity**. Shomron et al [1] purposed exploiting the **spatial correlation** inherent in CNNs and predict activation values, thus reducing the needed computations in the network. They introduced a heuristic that predicts which activations are zero-valued according to nearby activation values, in a scheme they call **cross-neuron prediction**.

In this work, we further generalize the work done by Shomron et al, with the following steps:

1. Implement a custom CNN layer which we dub a "**Spatial**" layer, allowing for a fast and efficient prediction statistics collection on any chosen CNN architecture.
2. Create a hierarchal framework that allows for the cross-neuron predictions on any generic CNN architecture and with any generic dataset chosen.
3. Implement the class structure into a few chosen networks: AlexNet, ResNet18, ResNet34
4. Implement a greedy optimization algorithm to choose the best Mask Configuration, allowing for a maximal number of saved MAC operations, under a certain loss in accuracy ε and based on changing optimization granularity.
   The algorithm was implemented in 4 modes: Max Granularity, Uniform Layer, Uniform Filter and Uniform Patch.
5. Test framework on the chosen networks with varying datasets

Steps 1-4 will be detailed in Sections 4 & 5 with the results displayed on Section 6.

1.1 Main Contribution

We bring forth a complete end-to-end framework enabling testing of cross-neuron predictions on any generic CNN network and dataset. We establish a "knob", controlling prediction performance vs accuracy degradation. We show promising results of an up to **27.15%** reduction in MAC operations on AlexNet and the CIFAR10 dataset, and up to **17.51%** on ResNet18 and the MNIST dataset, with **no decrease in accuracy**.

We further show on the AlexNet and CIFAR10 setup a possible **increase** in accuracy after retraining, while keeping the same amount of MAC operation savings. We demonstrate a **47.77%** decrease in MAC operations under a mere **0.5%** accuracy loss is achievable under retraining.

2. Introduction

In this section we will briefly introduce the various topics this project is about.

2.1 Convolution Neural Networks - Basics [2]

2.1.1 Overview

Convolutional Neural Network is a class of deep neural network that is used for Computer Vision or analyzing visual imagery.

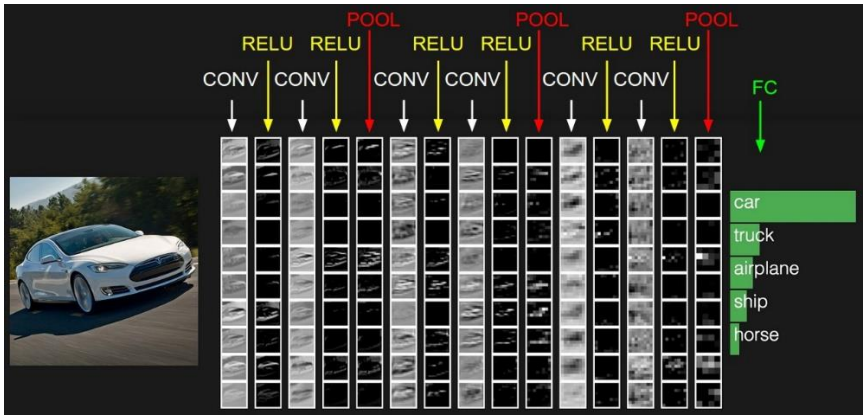

Figure 1: A basic overview of a simple CNN used for vehicle classification, taken from [3]

2.1.2 Convolutional Layer

Computers read images as pixels and it is expressed as matrix (NxNx3) — (height by width by depth). Images makes use of three channels (RGB), so that is why we have a depth of 3.

The Convolution Layer makes use of a set of learnable filters. A filter is used to detect the presence of specific features or patterns present in the original image (input). It is usually expressed as a matrix (MxMx3), with a smaller dimension but the same depth as the input file.

This filter is convolved (slid) across the width and height of the input file, and a dot product is computed to give an activation map.

Different filters which detect different features are convolved on the input file and a set of activation maps is generated and passed to the next layer in the CNN.

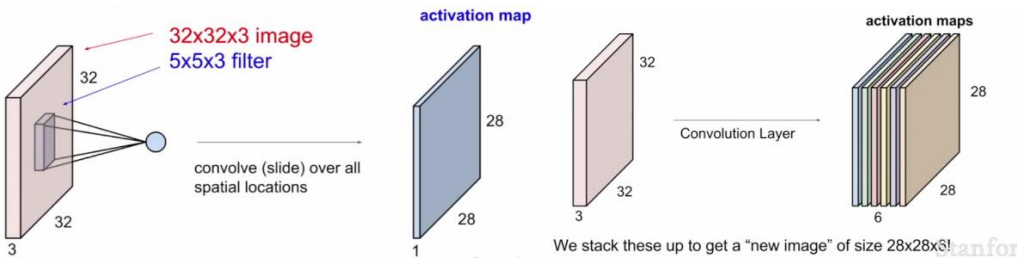

Figure 2: A basic view of input and output of a CNN layer, from [3]

2.1.3 Activation Function

Activation function is a node that is put at the end of or in between layers of neural networks. They determine if the neuron should fire or not.

> "The activation function is the nonlinear transformation that we do over the input signal. This transformed output is then sent to the next layer of neurons as input."—Analytics Vidhya

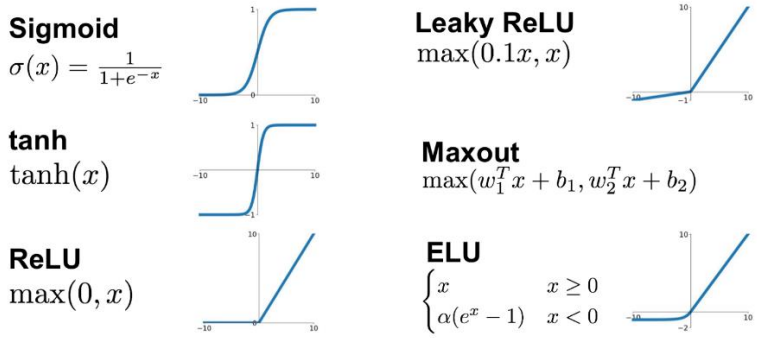

Figure 3: Possible activation functions

ReLU function is the most widely used activation function in neural networks today. One of the greatest advantage ReLU has over other activation functions is that it does not activate all neurons at the same time. As seen in Figure 3, the ReLU function converts all negative inputs to zero and the neuron does not get activated. Thus, it is very computational efficient as few neurons are activated per time. It does not saturate at the positive region. In practice, ReLU converges six times faster than hyperbolic tangent and sigmoid activation functions.

2.1.4 Fully Connected Layer (Dense layers)

In a dense layer, the neurons are fully connected to all the activations from the previous layers. Their activations can hence be computed with a matrix multiplication followed by a bias offset. This is the last phase for a CNN network.

The final output of the last dense layer is inserted into a k-Softmax probability function, allowing for a classification into k classes.

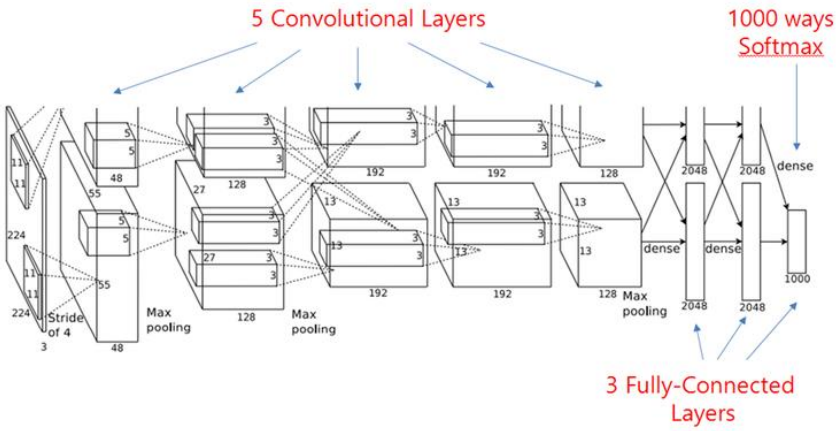

Figure 4: The AlexNet architecture, as shown in [2]

2.2 Convolution Neural Networks – Power Considerations

2.2.1 Field of View

Let us consider a simple CNN architecture for classifying digits, for example, on the well-known MNIST dataset. The propagation of data to a specific neuron in a given convolution layer is reliant only on some of the input pixels, but not all of them. Only the pixels that are involved in the actual convolution operation of that neuron are relevant – these stem directly from the size of the filter used. This result is demonstrated in source [4], allowing for clear visualization of the data propagation.

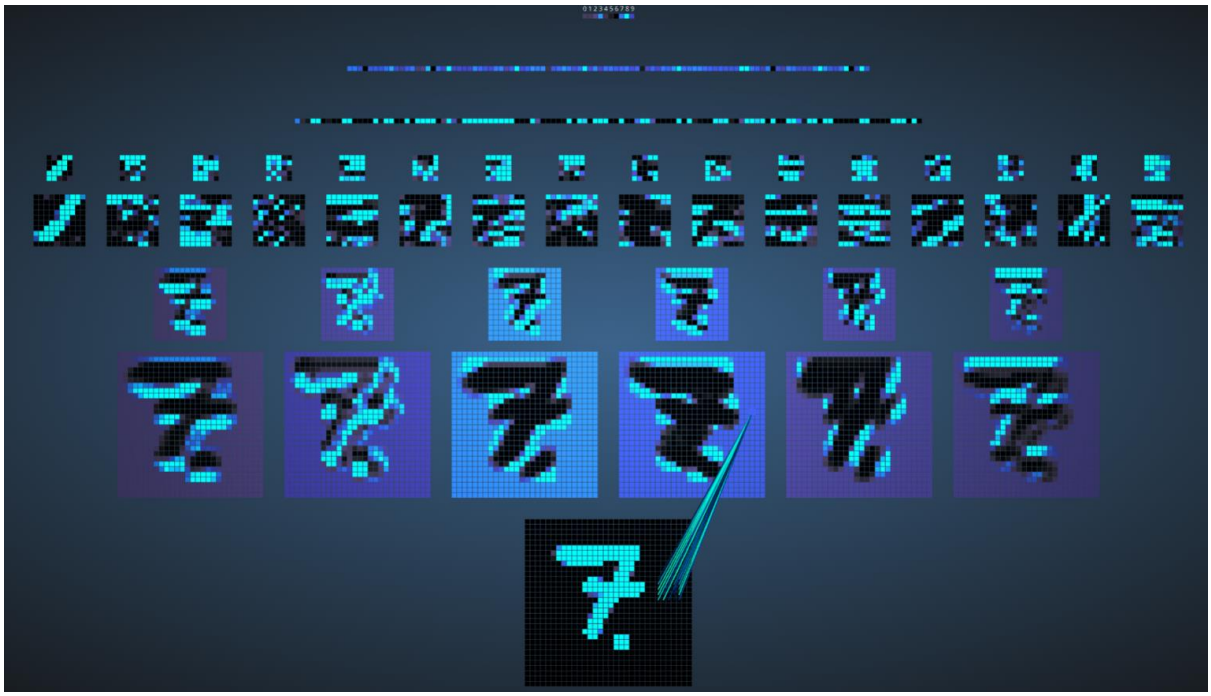

Figure 5: CNN output filter maps visualization on MNIST trained CNN classifier for the digit 7 [4]

2.2.2 MAC Operations and ReLU considerations

Consider a given filter i of a convolution layer j. The output of the neuron is calculated by:

$$\text{bias} + \sum_{k \in \text{Field of View}} x_k w_k^{ij}$$

For a filter of size 3x3 for example, we would calculate $3 \cdot 3 = 9$ multiply operations, and sum $3 \cdot 3 - 1 + 1 = 9$ times, thus resulting in 9 MAC (Multiply and Accumulate operations). As with any arithmetic computations, these cost us power. Notice that when this sum is negative, a ReLU activation on that layer truncates the sum to zero. Using this fact, we will try to predict when the convolution result will be non-positive and thus save MAC operations and power.

3. Project Specification

In this section we will briefly motivate this project and explain its targets.

3.1 Problem Description

Convolution neural networks have become a state-of-the-art approach to solving image related tasks. However, they suffer from being highly computationally intensive [5].

Predicting zero-valued activations has already been proven effective to reduce some of the computations [1] [5]. In this project we will optimize the prediction pattern to achieve the highest performance possible within a model's accuracy constraint.

Project Targets:

1. Learn how CNNs work
2. Learn how cross neuron activations prediction method works
3. Learn how Aklaghi et al. [5] optimized their solution
4. Optimize the mask patterns used in [1] to be more efficient, utilizing the optimization ideas from [5]
5. Evaluate prediction performance and model accuracy

3.2 Past Work Overview

In this section, we review the two main articles this project is based on.

3.2.1 SnaPEA [5]

In [5], Aklaghi et al proposed SnaPEA, a system for reducing computation-heavy convolution operations, exploiting the fact that the common ReLU activation layers truncate any results below zero. The offered two modes of optimization:

1. **Exact mode:** Assuring no loss in classification accuracy. This mode uses the observation that in the CNNs with ReLU activation layers, the inputs to the convolution layers are positive. Therefore, only performing the MAC operations with the negative weights can turn the convolution output negative. The exact mode adds additional logic on top of CNN layers, which reorders the weights of convolution kernels based on their sign such that the positive subset are followed by the negative subset. The reordering enables SnaPEA to first perform MAC with the positive subset and then cut the computation and apply activation function earlier in the case of observing a negative partial output during the computation with negative weights.
2. **Predictive mode**: Trades classification accuracy for larger savings. Speculatively cuts the computation short even earlier than exact mode. To control the accuracy, they develop a multi-variable optimization algorithm that thresholds the degree of speculation by exposing a knob to navigate the trade-offs between the classification accuracy and computation reduction. Thy implement a greedy

algorithm that outputs two parameters: a threshold and number of operations N. If the partial sum of the convolution crosses the threshold after N MAC operations the output of the ReLU layer is predicted as 0, otherwise the calculation proceeds as usual.

Results:

- SnaPEA in the exact mode, yields, on average 28% speedup and 16% energy reduction in various modern CNNs without affecting their classification accuracy.
- With 3% loss in classification accuracy, on average, 67.8% of the convolutional layers can operate in the predictive mode. The average speedup and energy saving of these layers are 2.02× and 1.89×, respectively. The benefits grow to a maximum of 3.59× speedup and 3.14× energy reduction.

3.2.2 Cross-Neuron Predictions [5]

To ease the computation intensity of CNNs, Shomron et al applies a common technique applied in GPPs – **value prediction**. They purpose a value prediction method which exploits the spatial correlation of activations inherent in CNNs. The arguments is as follows: neighboring activations in the CNN output feature maps (**ofmaps**) share close values (illustrated in the figure below).

Figure 6: Taken from [1], examples of AlexNet CONV3 ofmaps (after ReLU) for five arbitrary filter channels. Spatial correlation is evident. Black pixels are zero.

Therefore, some activation values may be predicted according to the values of **adjacent** activations. By predicting an ofmap activation, an entire convolution operation between the input feature map (ifmap) and the kernel may be saved.

Method:

They propose a prediction method by which zero-valued activations are predicted according to nearby zero-valued activations. First, ofmaps are divided into square, non-overlapping prediction patches. The ofmaps are padded with zeros so predictions can also be made in the presence of margins. Next, the activations positioned diagonally in each patch are calculated. If these activations are zero-valued, the remaining activations within that patch are predicted to be zero-valued as well, thereby saving their MAC operations. The figure below depicts this method, with 2x2 patches.

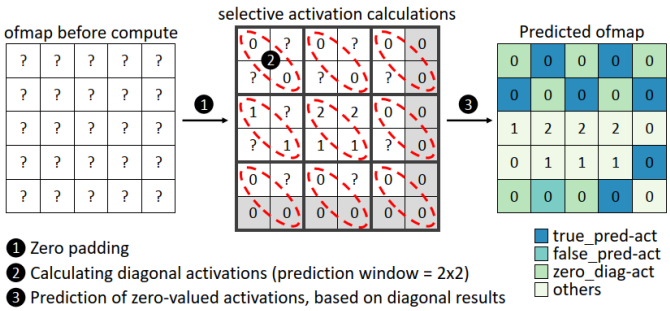

Figure 7: Taken from [1], Illustration of the prediction method

Results:

All configurations were run on the ImageNet dataset.

2x2 Patch:

- 34.8% MAC savings under a Top-1 degradation accuracy of 1.9% on the AlexNet network.
- 20.8% MAC savings under a Top-1 degradation accuracy of 11% on the ResNet18 network. After retraining the module with the proposed prediction method, MAC savings increased to 22.7% and degradation accuracy decreased to 2.7%.
- 30.8% MAC savings under a Top-1 degradation accuracy of 3.6% on the VGG-16 network.

3x3 Patch:

- 40.8% MAC savings under a Top-1 degradation accuracy of 4% on the AlexNet network.
- 23.5% MAC savings under a Top-1 degradation accuracy of 17.6% on the ResNet18 network.
- 36.2% MAC savings under a Top-1 degradation accuracy of 8.4% on the VGG-16 network.

In addition, retraining the AlexNet and the VGG-16 modules improved their accuracy. However, they suffered a slight degradation in MAC reduction, in contrast to the ResNet18 module.

4. Architecture

In this section we will review the architecture proposed in this project, and explain its function.

4.1 Spatial Layer

We build upon the work done by Shomron et al [1], and purpose a generalization of its idea. We propose a custom CNN layer which we dub a "**Spatial**" layer which computes the cross-neuron prediction for any given prediction mask. We recognize the fact that the flat prediction mask (**pmask**) used by Shomron et al may not be optimal, and other, more varied masks may achieve superior results, while supplying lower accuracy degradation. In general, for a patch of size pxp, may receive as many as $\Omega_p^* \triangleq 2^{p \cdot p} - 2$ possible **singleton** masks (relevant for a single patch in the ofmap), removing the degenerate all 1s or all 0s mask.

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Rightarrow \begin{matrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \\ & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & \\ \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} & \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} & \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} & \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} & \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \end{matrix}$$

Figure 8: The 2x2 prediction mask used by Shomron et al (left), versus all possible masks (right). The degenerate all ones or all zeros masks were removed, seeing they define deterministic and degenerate predictions

For a given mask, the activations positioned with "1" in each patch are calculated. If these activations are zero-valued, the remaining activations within that patch are predicted to be zero-valued as well, thereby saving their MAC operations.

This custom layer was implemented with Python utilizing the well-known PyTorch module, and runs efficiently with a GPU compatible implementation. The layer receives a given **full** pmask (combined from singleton patch-size pmasks, targeted to be in the shape of the ofmap), and collects statistics such as the number of MAC operations saved via the supplied pmask.

4.2 Spatial Network Framework

Recognizing the difference between different network architectures, such as VGG16 or AlexNet, we model a new class dubbed "**SpatialNet**". This class inherits from the PyTorch neural network class, and implements generic functions that deal with the control and statistical analysis of all Spatial layers placed inside the network's architecture definition. This allows an easy and simple way to implement Spatial layers into any desired CNN. We implement the framework on the **AlexNet, ResNet18** and **ResNet34** classes.

5. Optimization

In this section will detail the optimization algorithm and its various control parameters

5.1 Overview

The main idea of this project is as follows:

Given a desired SpatialNet classification network and chosen dataset, along with an allowed parameter ε detailing the maximal accuracy loss allowed, find the optimal set of prediction masks, allowing for the maximal amount of MAC operations saved, with under ε classification accuracy loss.

The steps are as follows:

1. Insert Spatial layers after any chosen Convolution layers with ReLU activations
2. Vanilla train the network on the dataset with the spatial layers disabled. Save baseline 0-1 classification accuracy.
3. Configure the chosen optimization hyper-parameters, choosing the desired mode of optimization.
4. Using the trained model, run the optimization algorithm, looking for the best set of full prediction masks with under ε classification accuracy loss from the baseline accuracy. We note that for every spatial layer in the architecture, exists an optimal prediction mask.

5.2 Optimization Algorithm

5.2.1 Hyper-parameters

We allow for a full configuration of many hyper-parameters, allowing for greater freedom of optimization and analysis. In this section we will describe these hyper-parameters, which are the input of our optimization algorithm.

5.2.1.1 Network Choice

As noted in [1], some network architectures supply sparser ofmaps than others. We tested two different networks – ResNet18 [7] and AlexNet [8]. AlexNet has five convolution layers, therefore out setup has five spatial layers. Similarly, the Resnet18 setup has 17 spatial layers. For the ResNet18 architecture we introduced two optimization settings – regular and UniBlock. In the regular setting, each layer is defined and optimized separately. In the UniBlock setting, we have a **coupled** set of layers that are optimized jointly (for example, layers 1-4 have the same prediction mask).

5.2.1.2 Dataset Choice

We implemented many different dataset choices into the framework, including ImageNet, TinyImageNet, STL10, MNIST, CIFAR10, and FashionMNIST. Dataset choice is a primary factor when looking at computation runtime. Where the ImageNet dataset contains over 1,000,000 RGB photos with an average size of 256x256, the CIFAR10 dataset contains only 60,000

grayscale photos of size 32x32. These factors decide the forward runtime of the network, which is the most substantial part of the algorithm's runtime. Therefore, we can approximate the total runtime of our optimization algorithm to be: $T \approx \Omega \cdot F$ where Ω is the total number of options checked, and F is the runtime of a single forward run on our test dataset. In addition, we have a test set size configuration parameter, i.e. it is possible to run the optimization algorithm on a part of a dataset. In the results shown in section 6, this parameter was set to one thousand.

5.2.1.3 Maximal Allowed Accuracy loss – ε

The ε parameter denotes the maximal acceptable accuracy loss during the optimization process

5.2.1.4 Optimization Mode

Our optimization algorithm runs in four different modes, detailed in Figure 9 below.

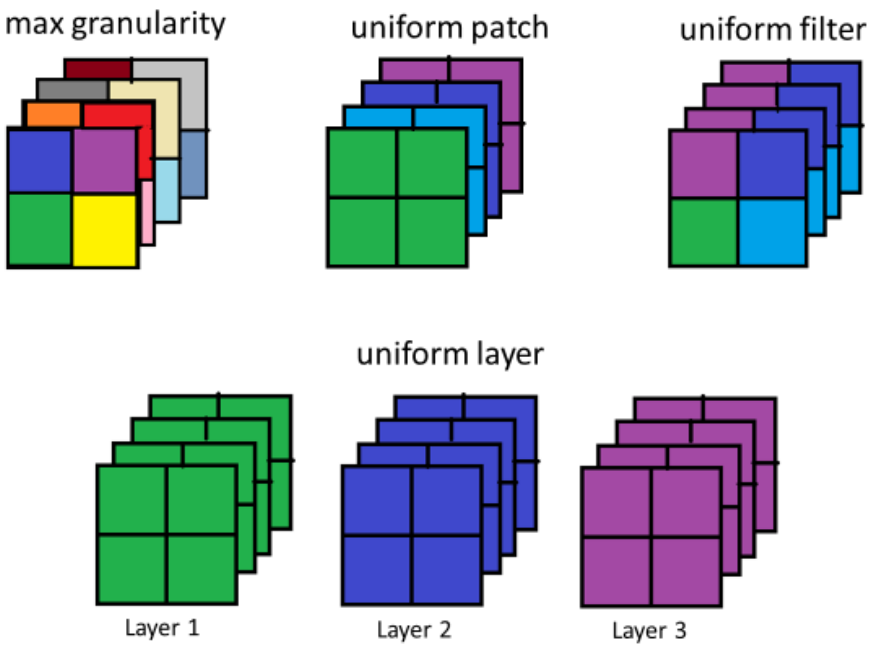

Figure 9: Implemented optimization modes, detailed with a rough sketch. Each square is a kernel ofmap mask, and each set of squares is a full ofmap for a given spatial layer. Different colors relate to different singleton pmasks

For each mode of operation, we define its optimization target as the following:

- Max Granularity: Targets the **patch** level
- Uniform Filter: Targets the **patch level across the layer** (all channels within a single layer)
- Uniform Patch: Targets the **channel** level
- Uniform Layer: Targets the spatial **layer** level

The key presumption behind the algorithm is that targets are **uncorrelated**. Given N targets, optimizing jointly for all targets is equivalent to N sole optimization tasks of each. This presumption is clearly not withheld, seeing ofmaps from deep layers are derived directly from ofmaps of shallower layers in the architecture, but it allows us to deal with the combinatorial explosion, making the optimization feasible. We roughly describe each optimization mode:

5.2.1.5 Patch Size (p parameter)

The singleton size of the estimator block. In the baseline case (Diagonal in [1]) for a value of p=2 we would receive the singleton pmask:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

For a patch size of 3 in the baseline case, this would amount to:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

We note that increasing the prediction patch size presents a tradeoff. On the one hand, a large patch size increases the number of activations that may be predicted per patch, thus allowing for increased operations saved. On the other hand, spatial correlation diminishes as the size of the prediction patch increases, which: (1) decreases the number of prediction patches inside the entire layer pmask; and (2) increases the false prediction rate, since the area around the with non-1s possible increases. These false predictions will decrease the model accuracy.

5.2.1.6 One's Range or Patterns

The optimization algorithm requires as input possible patterns for a single pmask, where pmask is a binary mask of size p x p. The optimization algorithm will build from these patterns a complete prediction mask for the whole network that maximizes operations saved and does not exceed the maximal allowed drop in accuracy.

There are two options for suppling these possible pmasks. The first is through a patterns parameter, which is a 3D array that contains all the patterns to choose from during the optimization process. We denote this set of patterns as Ω_p^*. The second option is to provide a one's range. This parameter controls the number of 1s we would like to see in the pmask, and is supplied as a range between 1 and p^2-1. All possible pmasks with number of "1s" in this range are generated and the optimization process continues the same way as in the patterns option.

We note that there is an inherent tradeoff between different masks – The more "0s" supplied with the mask – the higher the chance to save MAC operations, and the more operations that are saved. With this, the higher the chance to predict wrongly, thus causing possible classification accuracy degradation.

The results presented in section 6 all use the one's range option to generate possible patterns for optimization.

5.2.1.7 Granularity Threshold Parameter

We note that when optimizing in the max granularity mode, we test the effect of a small p x p patch on the accuracy of the entire network. Seeing this is a tiny change when compared to the entirety of the network, we look for some way to strengthen its effect. We increase the spread of each target checked to be something more substantial, by increasing its area of effect. For a given target pmask choice, we define the GT parameter. This parameter clusters close by patches and defines them as a single target for optimization. When GT is lowest, the whole kernel would be a single patch, and we receive the uniform patch mode. The GT parameter has effect only in the max granularity mode.

5.2.2 Combinatorial Troubles

We begin with a worrying problem – the combinatorial explosion expected when looking for the optimal mask. We note the patch size hyper-parameter that controls the number of combinatorial options to choose from (detailed in section 5.2.4.1). Notice that while taking into account all valid mask options, the growth in the options for a singleton pmask options is exponential:

$$\Omega_p^* \triangleq 2^{p^2} - 2 = \begin{cases} 2^4 - 2 & p = 2 \\ 2^9 - 2 & p = 3 \\ 2^{16} - 2 & p = 4 \end{cases} = \begin{cases} 14 & p = 2 \\ 510 & p = 3 \\ 65534 & p = 4 \end{cases}$$

This is only the beginning of the problem, seeing that multiple singleton pmasks are to be inserted into the creation of a full pmask, relevant for a single spatial layer. We explain the combinatorial calculation with an example:

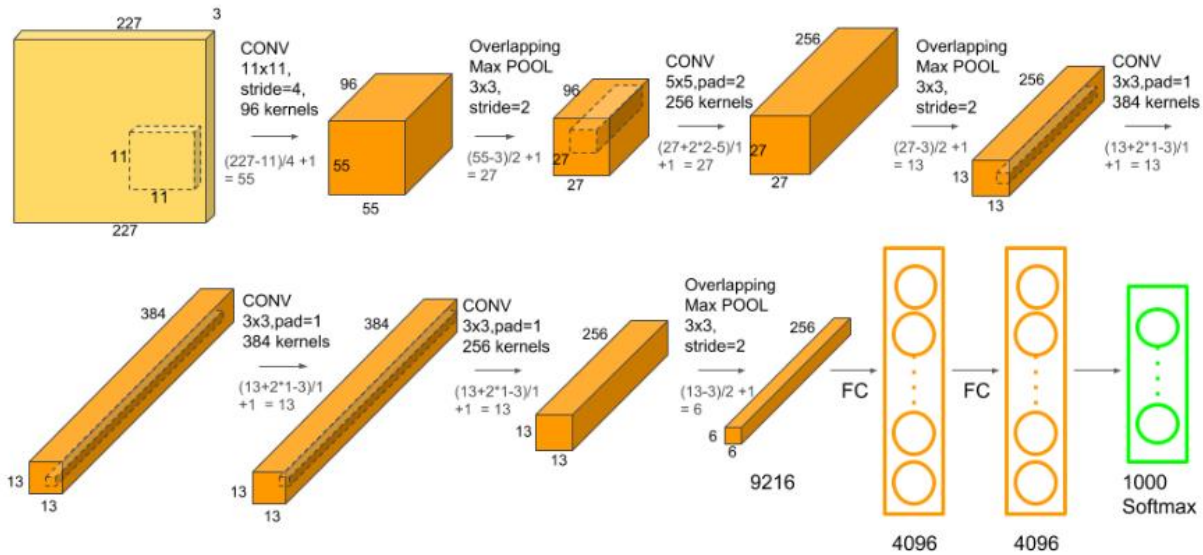

Figure 10: AlexNet ofmaps for input of size 227x227 as detailed in [6]

Given an input of size 227x227, the AlexNet architecture has 96 kernels/filters of size 11x11. This creates a 96x55x55 ofmap, which will then be inserted into the Spatial layer. If we choose p=2, we need to split this map into 2x2 blocks. We therefore pad the map with a section of zeros, resulting in an ofmap of size 96x56x56. We can now insert:

$$N = \frac{96 \cdot 56 \cdot 56}{2 \cdot 2} = 75,264$$

Where N is the number of different patches. We now have the task of choosing which of the $\Omega^*_{p=2} = 14$ options for singleton masks is best for each patch, resulting in:

$$\Omega = 14^N = 14^{75264}$$

Where Ω is the number of options for a **single** full pmask, and this is only for a single spatial layer.

Note that for the AlexNet architecture, as detailed [6], if we place a spatial layer after each convolution layer, a total of

$$N = \frac{96 \cdot 56 \cdot 56}{2 \cdot 2} + \frac{96 \cdot 28 \cdot 28}{2 \cdot 2} + \frac{256 \cdot 28 \cdot 28}{2 \cdot 2} + 2 \cdot 384 + 2 \cdot 256 \frac{14 \cdot 14}{2 \cdot 2} = 206976$$

Where N is the number of different slots for singleton pmask placement are possible, creating $\Omega = 14^{206976}$ different options.

If we optimize by this granularity utilizing an exact exhaustive search, with each forward run needed to compute the number of operations saved and the loss in accuracy, taking 1msec, it would many more years than the number of atoms in the known universe to finish the computation.

5.2.3 Greedy Optimization

The problem of optimizing over the mask set is therefore most likely NP-Hard, and exponential in the required runtime complexity required to choose the best possible mask configuration for the entire network, thus unfeasible computationally.

We borrow ideas from [5] and design a multilevel greedy optimization algorithm, running in four possible modes detailed in 5.2.1.4.

5.2.3.1 Exact solution under the Uniform Layer mode

We would like to work with a higher granularity, reducing combinatorial troubles. For the Uniform Layer setup, we define the following scheme:

Given a set of singleton pmasks Ω, construct a **uniform** full pmask, where all patches originate from the same singleton pmask. This means that for a given layer, there are Ω^*_p options to test. The number of combinatorial options to test is therefore:

$$\Omega = L^{\Omega^*_p}$$

Where L is the number of spatial layers in the network architecture. For L=5 and p=2 for example, we receive:

$$\Omega = 5^{14} \approx 6.1 \cdot 10^9$$

This is the only mode where an exact solution may be computed, under the assumption of a very small L and p.

5.2.3.2 Block Diagram

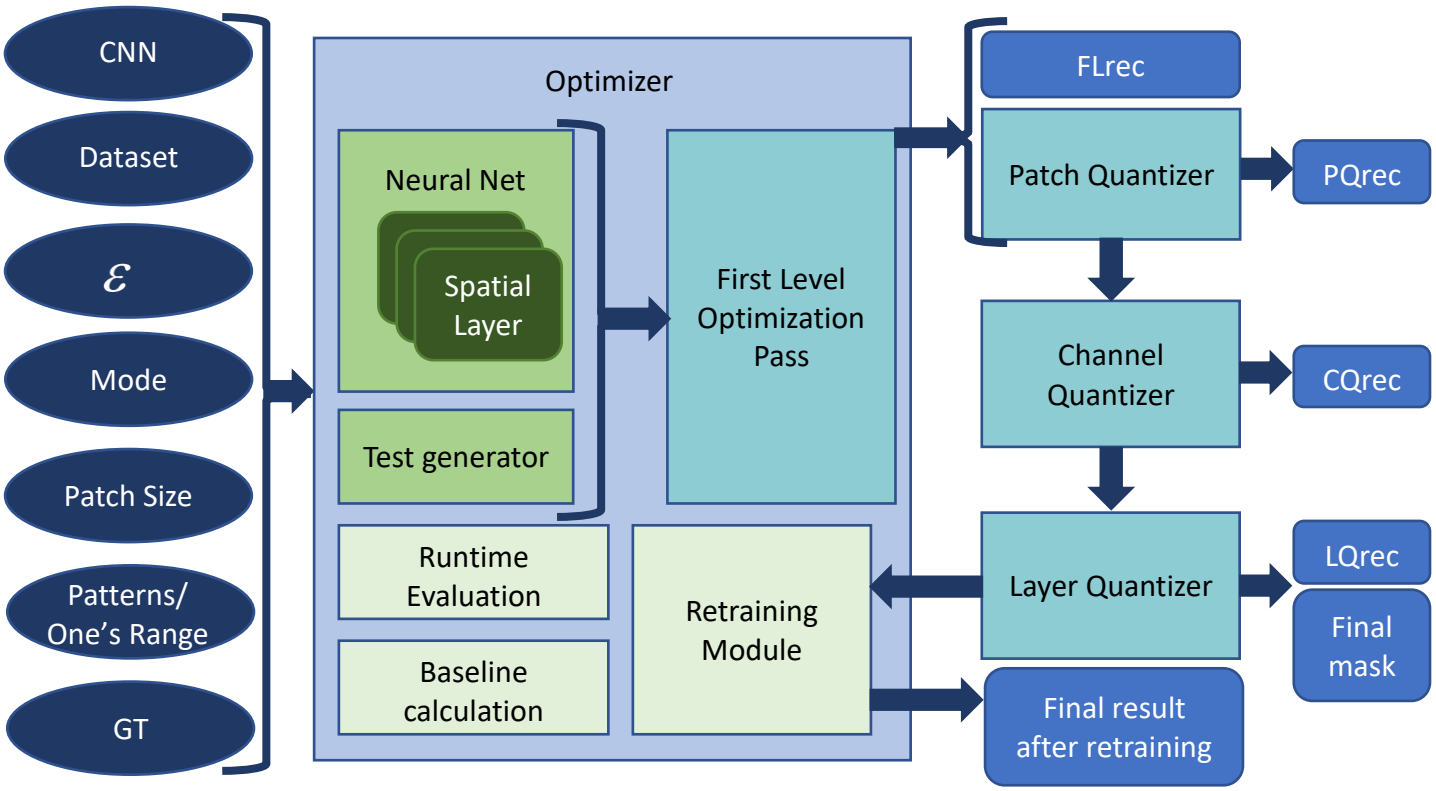

Figure 11: Block diagram of the implemented greedy optimization algorithm

Figure 11 depicts the software flow of our greedy optimization algorithm. The inputs discussed in section 5.2.1 are shown on the left side. From these inputs the optimizer class creates a neural network with spatial layers (that are discussed in section 4) and a test generator which are used to simulate the effects of different predictions masks on the accuracy and percentage of saved MAC operations. The optimizer class contains a baseline calculation module that generates baseline results to compare our optimization method to the method described at [1] (baseline i.e. a uniform network with diagonal pmasks). The runtime evaluation module calculates and displays the worst case runtime of our optimization algorithm for the four possible modes. An example of the runtime evaluation module's output can be found in section 11. The modules colored in teal in Figure 11, first level optimization pass and the three quantizers, are part of the optimization logic. They are described in the following four sections and in pseudo code in section 10. Each stage of the optimization process stores results in files for future optimization runs. These files appear in Figure 11 as rectangles with rounded corners (such as FLrec – first level record). After the optimization process that concludes in the layer quantizer block, the final prediction mask is stored and there is an option to use the retraining module to improve the received accuracy and operations saved with the final mask.

5.2.3.3 Greedy Algorithm under the Uniform Layer Mode

For each layer, set one of the Ω_p^* options, and run a full classification forward run through the system. Compute the number of operations saved, along with the relevant 0-1 classification accuracy. Collect all results, resulting in an array of: $L \times \Omega_p^*$ where each cell holds a 0-1 accuracy

and operations saved. This is the description of the FirstLevelOptimizationPass method described in the pseudo code in 10.3.

Then we preform the LayerQuantizerPass (detailed in 10.6) where we choose the best option for each layer, with respect to operation savings, and define it as the global pmask for the network.

Under the no correlation assumption, this is assured the best configuration. We have a possible problem of possibly not withholding the accuracy ε requested – this is handled by an iterative scheme to choose less accuracy-expensive pmask options, at the cost of lowering our operation savings, and is described in 10.7.

Note that the number of forward runs needed is a mere: $\Omega \simeq 2L \cdot \Omega_p^*$, which is vastly less than the numbers shown above, but also means that we are strictly looking at a much small subspace of possible options, most likely resulting in a sub-optimal solution.

5.2.3.4 Greedy Algorithm under the Uniform Patch Mode

Firstly, we perform FirstLevelOptimizationPass as in the previous section. In the uniform patch mode preforming the FirstLevelOptimizationPass entails setting one of the Ω_p^* options for each channel in each layer.

The results from the FirstLevelOptimizationPass are stored and passed to the channel quantizer. There, these results are filtered (results that surpass the maximal allowed accuracy loss are dropped) and sorted for each channel according to the number of saved operations. From the sorted channel results the channel quantizer builds layers in the following method: for a specific layer, we take the best channel option for all channels. Then, for the same layer, we take the second best channel option for all channels. Afterwards, the third best option is taken etc. Since not all channels in a layer may have the same number of options, the surplus is padded with an all "1s" pmasks. In this way, options for entire layers are built and a forward run is performed foe each layer option. The results of the channel quantizer forward runs are stored in CQrec (depicted in Figure 11) and passed to the layer quantizer for the next step in the optimization process. There, the best global prediction mask is chosen in the same way described in 5.2.3.3.

5.2.3.5 Greedy Algorithm under the Uniform Filters Mode

Again, we start with the FirstLevelOptimizationPass which entails setting one of the Ω_p^* options for each patch across all the channels in a layer. The results are stored and passed to the patch quantizer. The patch quantizer sorts the results according to the number of saved operations and filters options that surpass the allowed accuracy loss. From the sorted results, possible channels are built in the same method described in 5.2.3.4 for the channel quantizer, i.e. firstly the best options for all patches is chosen, the second best option for all patches, etc.

Since all the channels in a layer in uniform filters mode are identical, the channel quantizer stage is skipped. We move straight to the layer quantizer stage described in 5.2.3.3.

5.2.3.6 Greedy Algorithm under the Max Granularity Mode

The max granularity mode is similar to the uniform filters mode, described in the previous section 5.2.3.5, with two differences. Firstly, the FirstLevelOptimizationPass sets one of the Ω_p^*

options for each patch for each channel and for each layer (and not across all channels in the layer as in uniform filter mode). Secondly, the channel quantizer stage, described in section 5.2.3.4, is not skipped. Otherwise, this mode is similar to the uniform filters mode.

6. Results

In this section we describe various tests, and display some of the resulting prediction masks.

6.1 ResNet18, CIFAR10, 17 Spatial Layers

Network was trained to a 93.5% 1-0 classification accuracy on CIFAR10 (10 classes). We test our various optimization modes in contrast to the baseline result achieved in [1] – a uniform network diagonal mask. We receive a 3.5% decrease in accuracy, with 23.5% MAC operations saved for this configuration. GT parameter is set to a constant 10, Patch size is set to 2, and one's range parameter to [1, 2] (inclusive range)

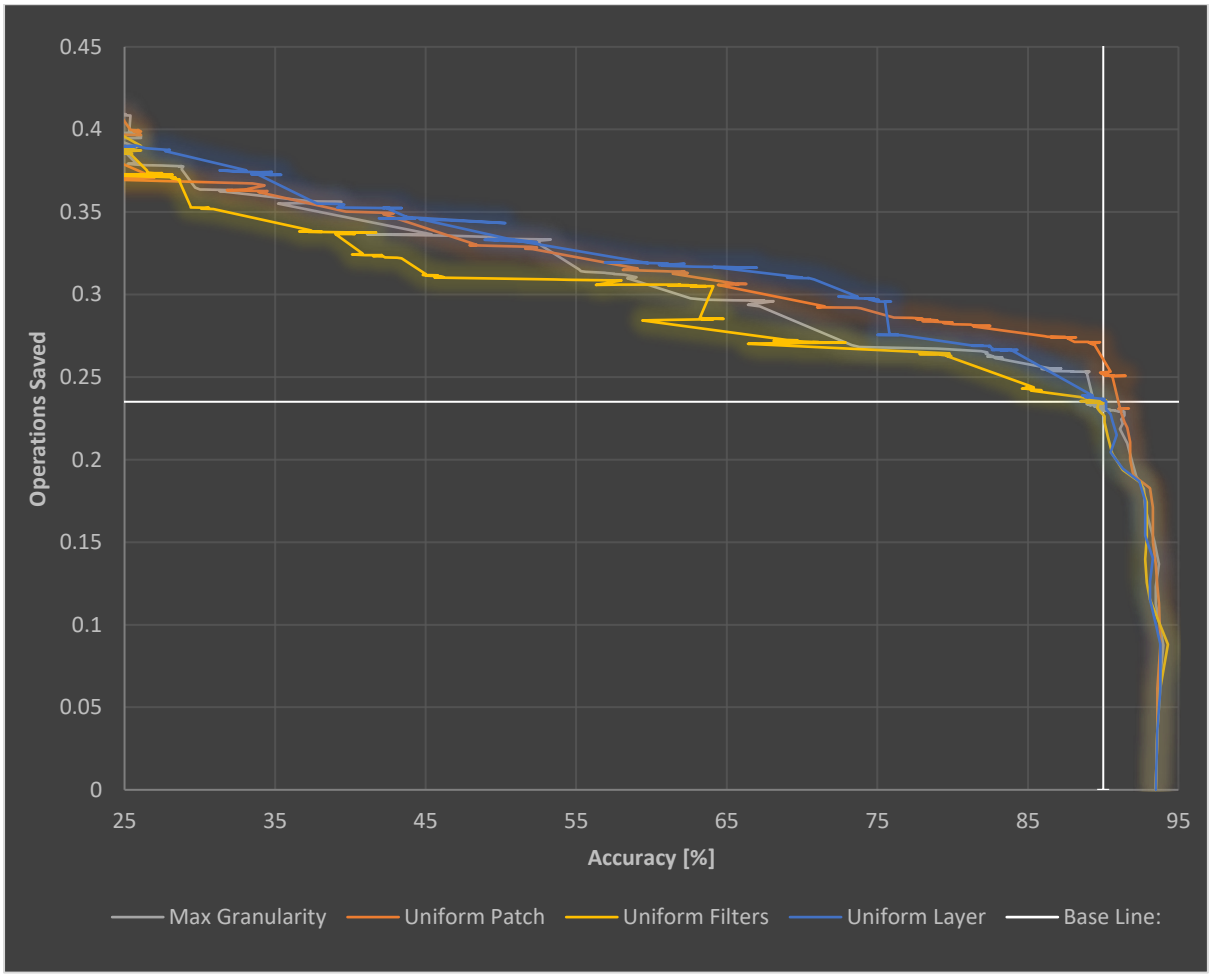

Figure 12: mode breakdown for Resnet18, CIFAR10, Patch Size = 2 and Ones Range = [1, 2]. The figure shows accuracy vs. operations saved for all the options generated in last stage of the optimization (the layer quantizer stage)

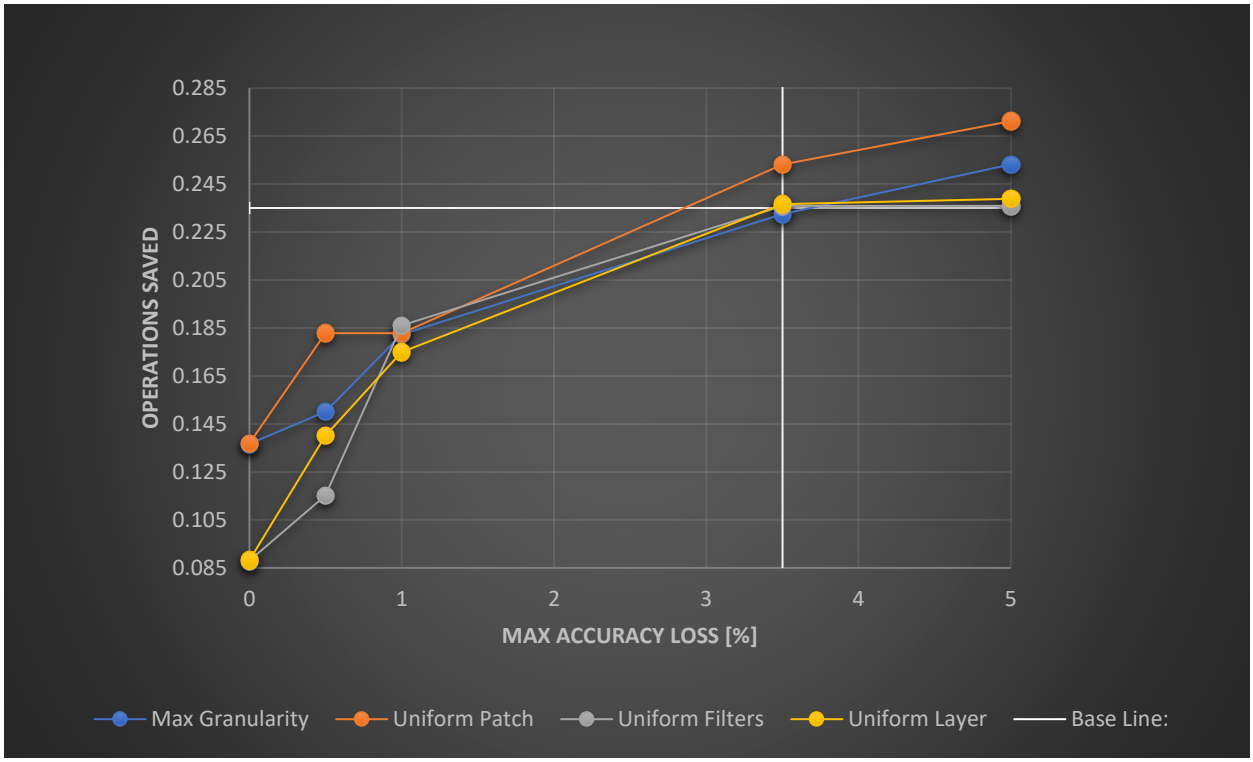

Figure 13: Operations saved for chosen values of maximal accuracy loss for Resnet18, CIFAR10, patch Size = 2, and Ones Range = [1,2]

6.1.1.1 Layer by Layer Breakdown of Saved Operations with the Final Mask

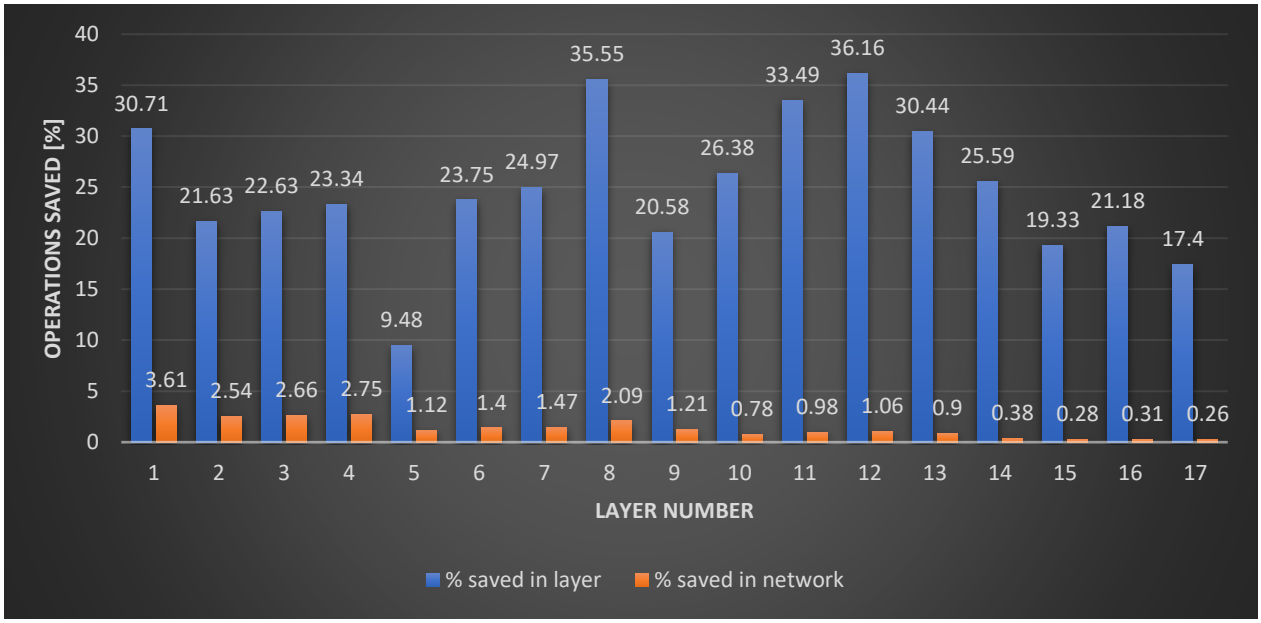

Figure 14: Operations saved for each layer in the final mask for uniform layers mode.

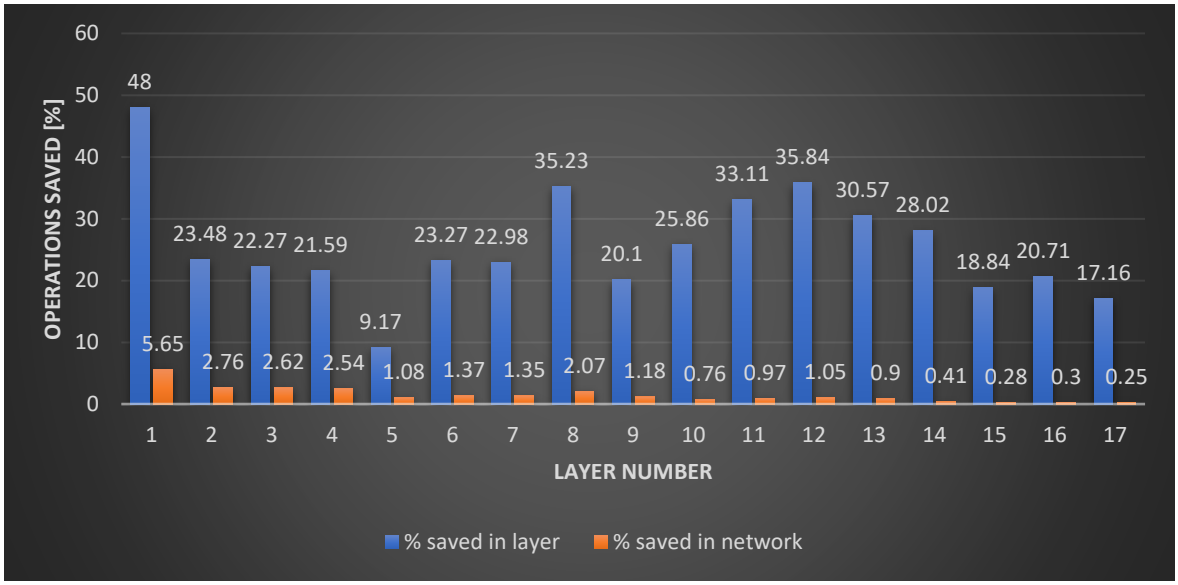

Figure 15: Operations saved for each layer in the final mask for uniform patch mode.

6.1.1.2 Final Mask Views

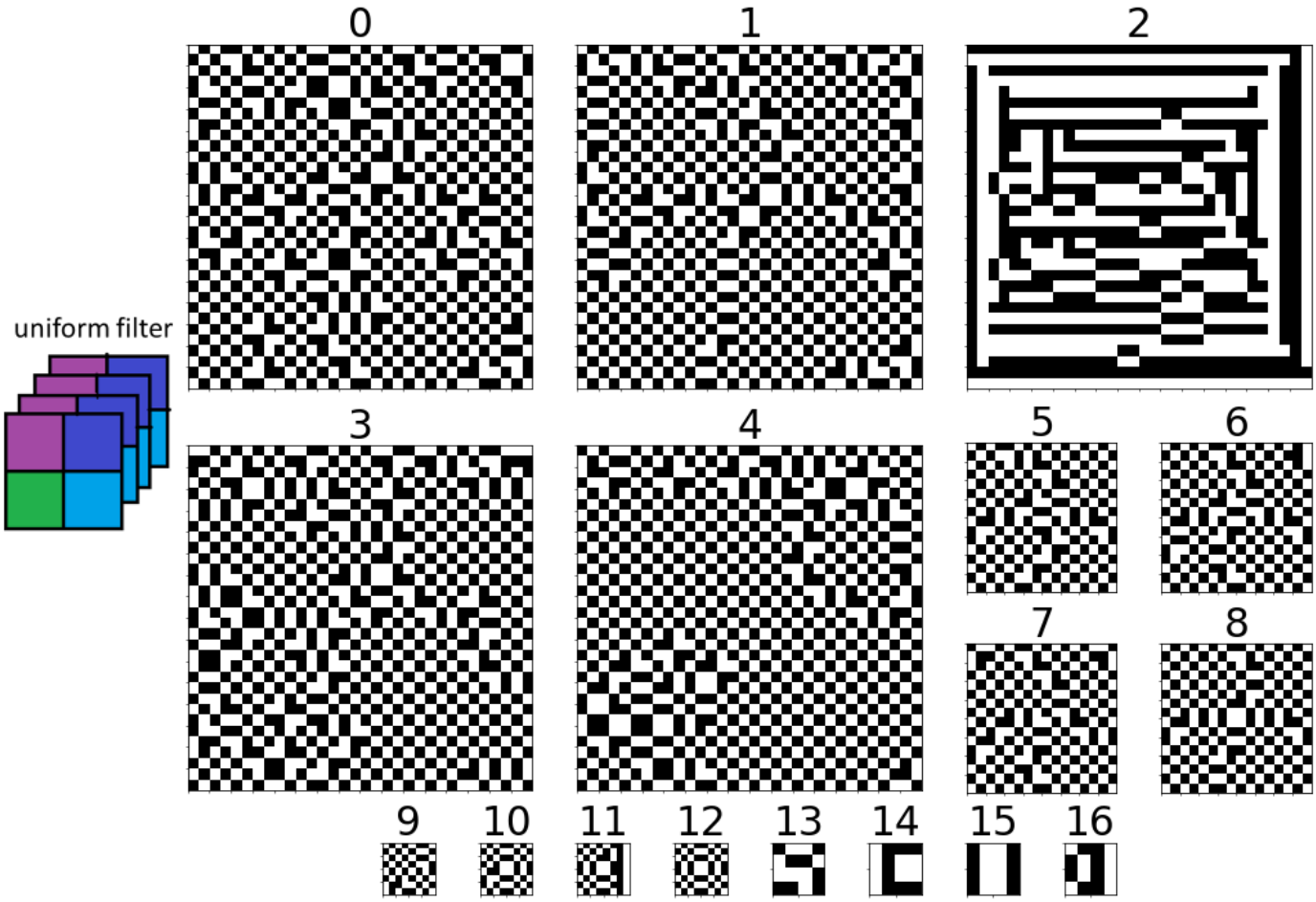

Figure 16: final mask view for uniform filters mode, Resnet18, CIFAR10, Patch Size = 2 and Ones Range = [1, 2].

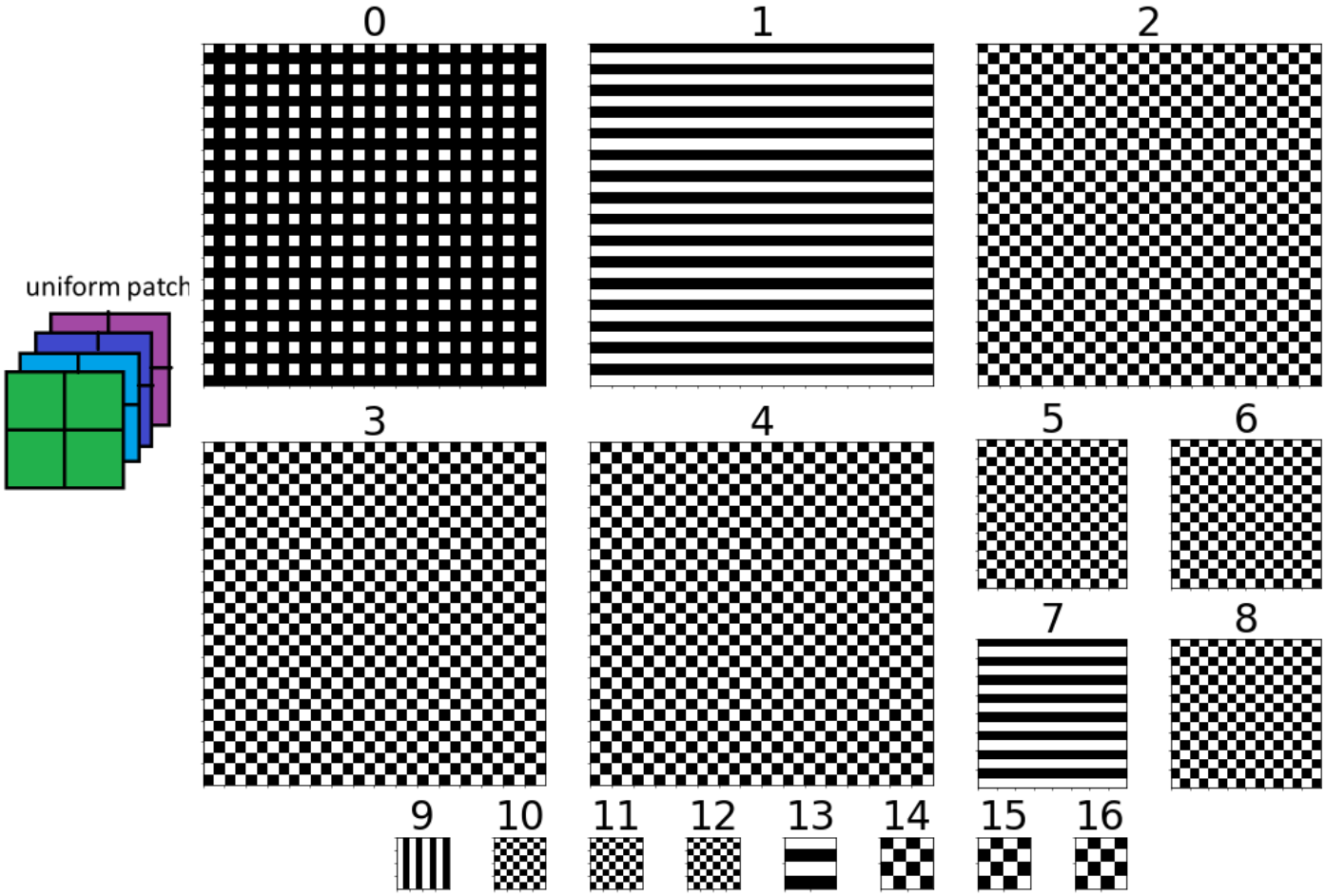

Figure 17: final mask view for uniform patch mode, Resnet18, CIFAR10, Patch Size = 2 and Ones Range = [1, 2]
Only the first channel of each layer is shown

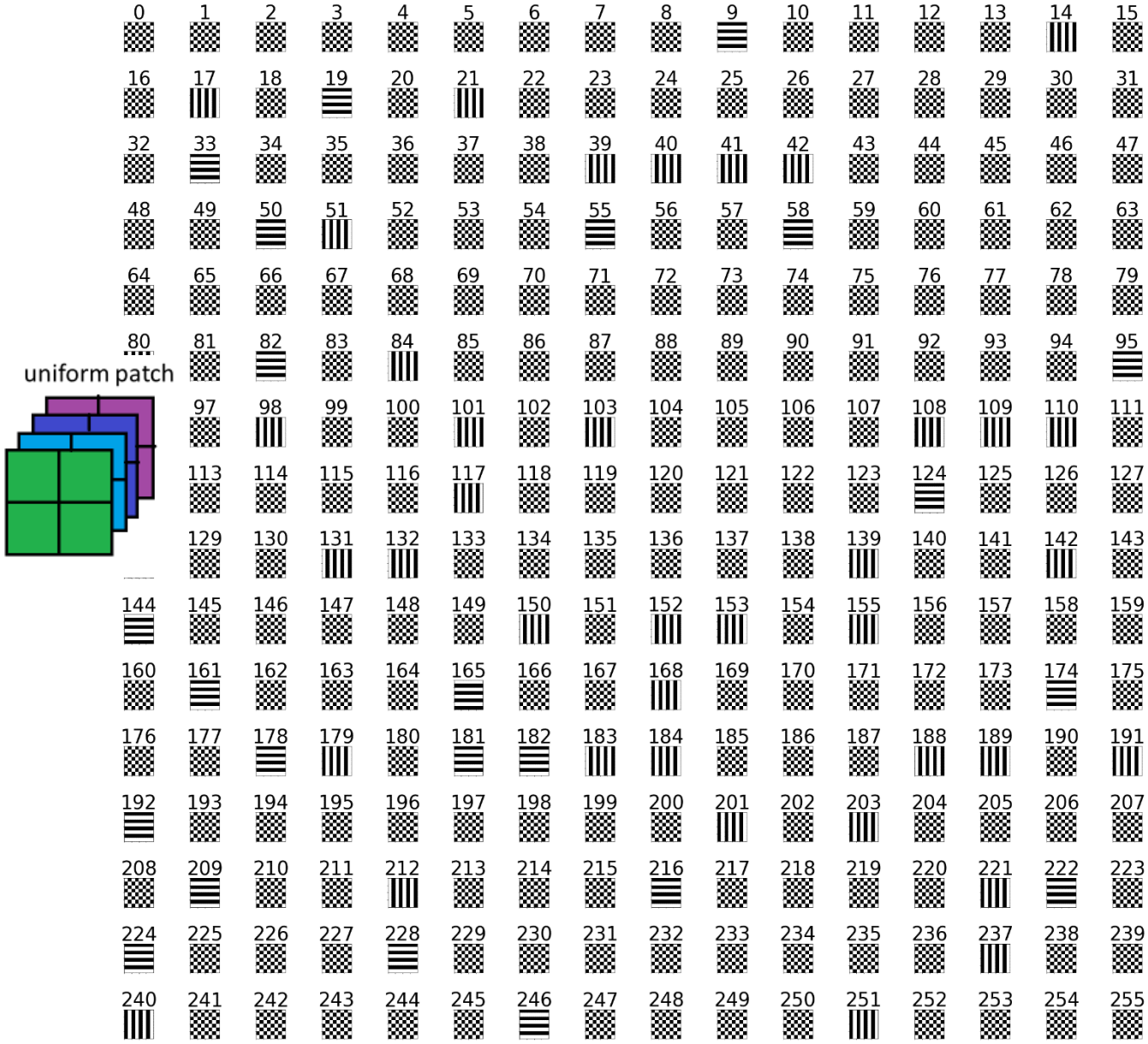

Figure 18: Channels of layer 10 in the final mask for uniform patch optimization, Resnet18, CIFAR10, Patch Size = 2 and Ones Range = [1, 2]

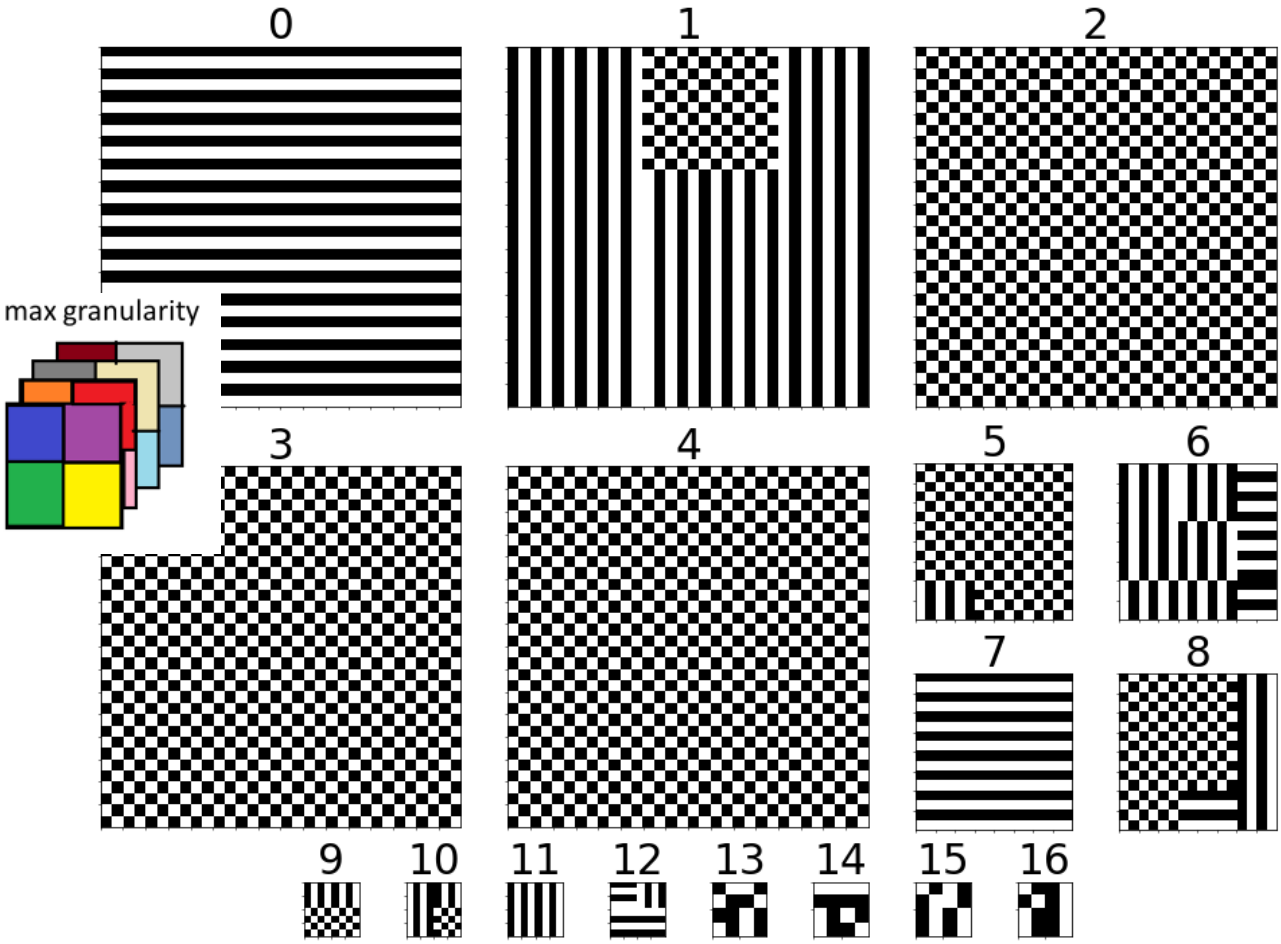

Figure 19: final mask view for max granularity mode, Resnet18, CIFAR10, Patch Size = 2, Ones Range = [1, 2] and GT=10
Only the first channel of each layer is shown

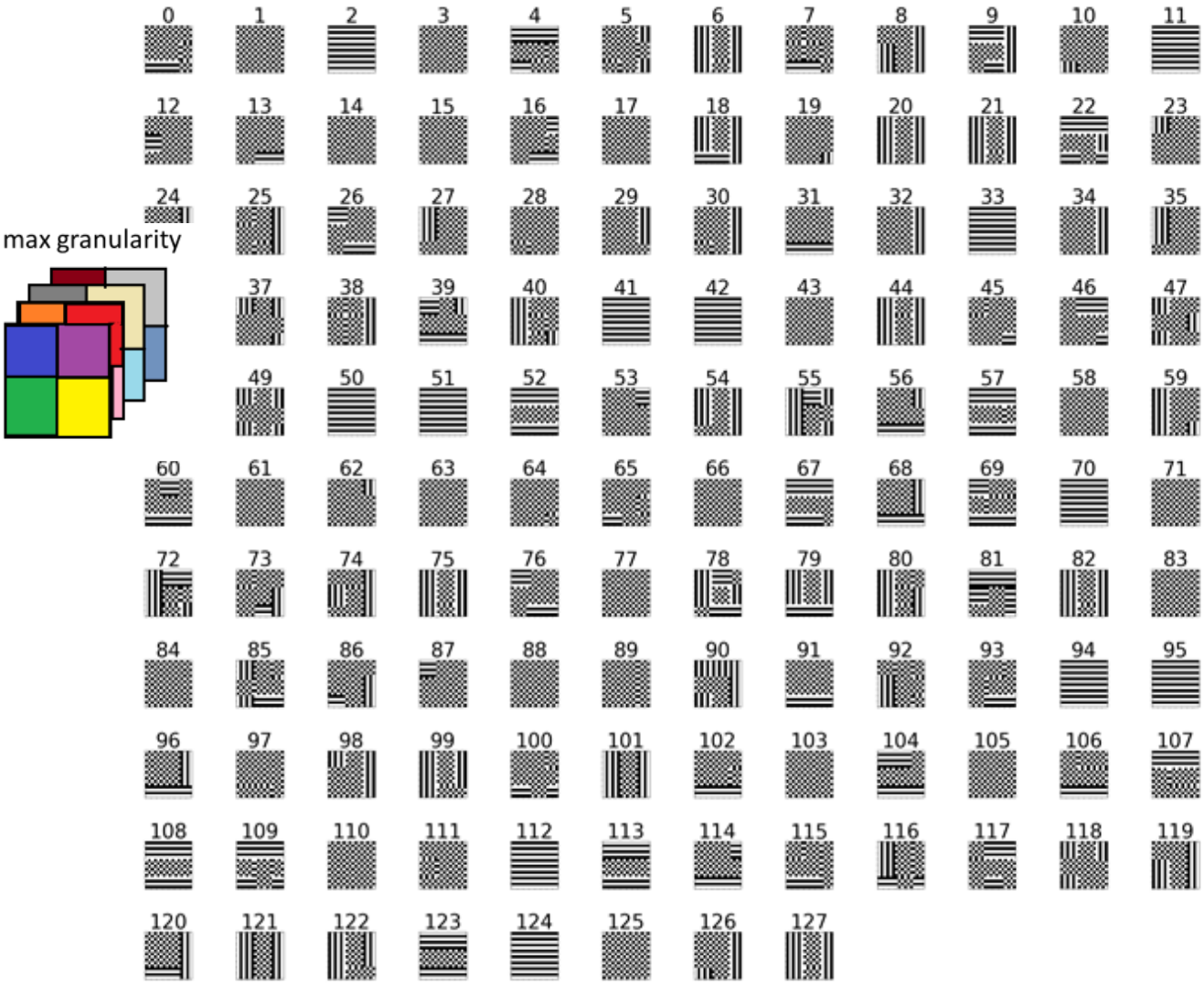

Figure 20: Channels of layer 7 in the final mask for max granularity optimization, Resnet18, CIFAR10, Patch Size = 2, One's Range = [1, 2] and GT=10

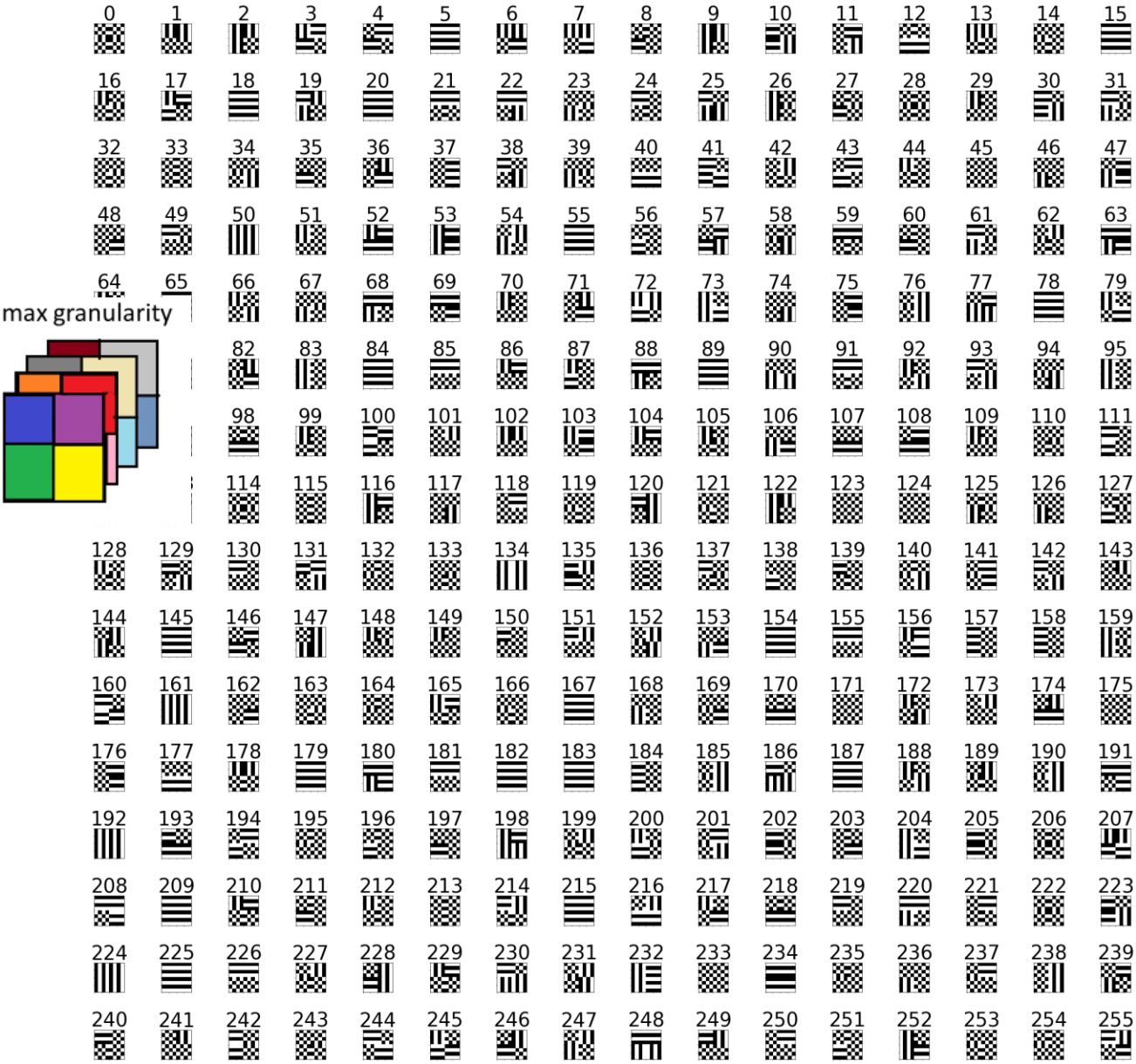

Figure 21: Channels of layer 10 in the final mask for max granularity optimization, Resnet18, CIFAR10, Patch Size = 2, One's Range = [1, 2] and GT=10

6.2 ResNet18, MNIST, 17 Spatial Layers

Network was trained to a 98.7% 1-0 classification accuracy on MNIST (10 classes). We test our various optimization modes in contrast to the baseline result achieved in [1] – a uniform network diagonal mask. We receive a 1.2% decrease in accuracy, with 16.76% MAC operations saved for this configuration. GT parameter is set to a constant 10. Patch size is set to 2, and one's range parameter to [1, 2] (inclusive range)

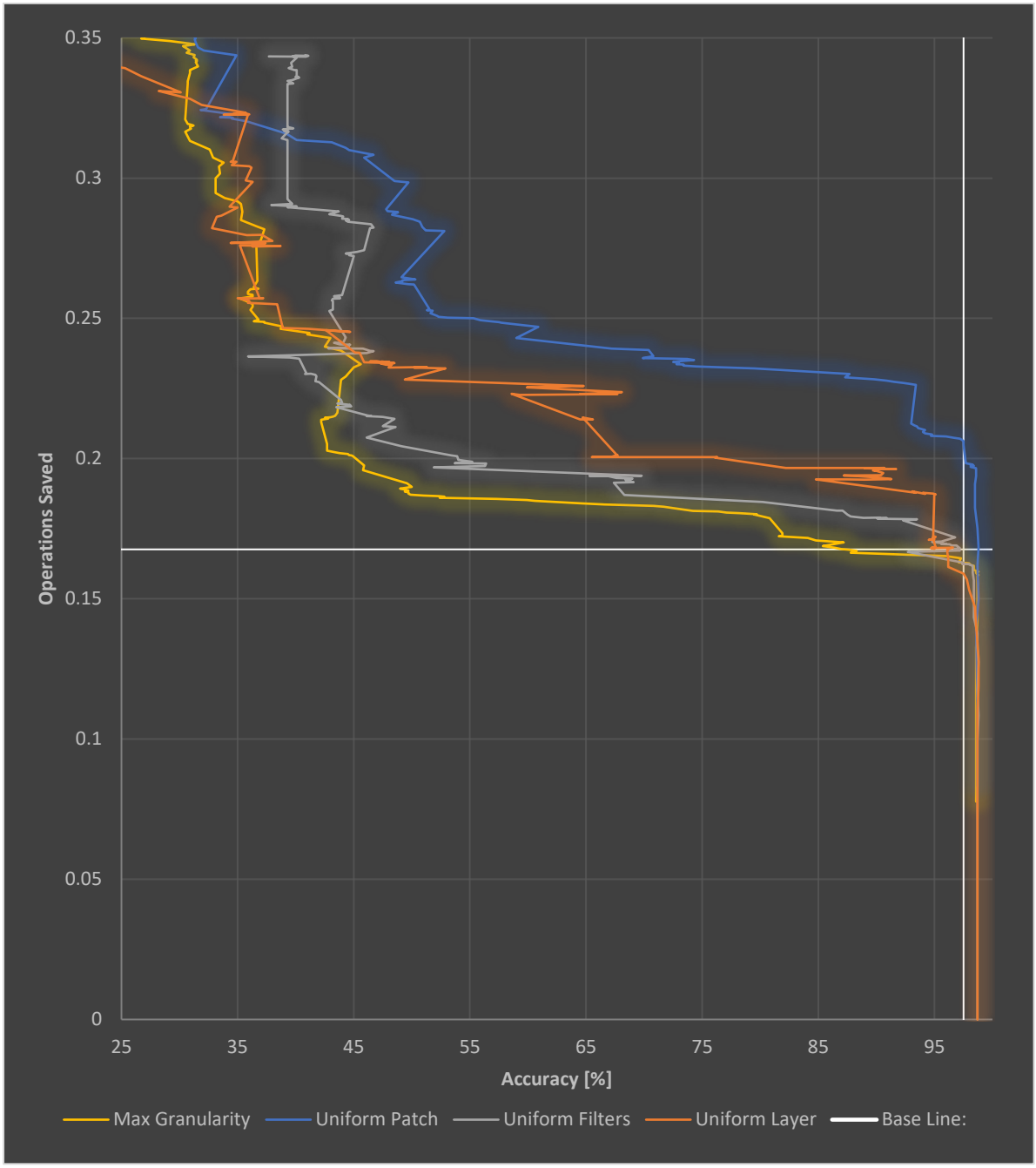

Figure 22: mode breakdown for Resnet18, MNIST, Patch Size = 2 and Ones Range = [1, 2]. The figure shows accuracy vs. operations saved for all the options generated in last stage of the optimization (the layer quantizer stage)

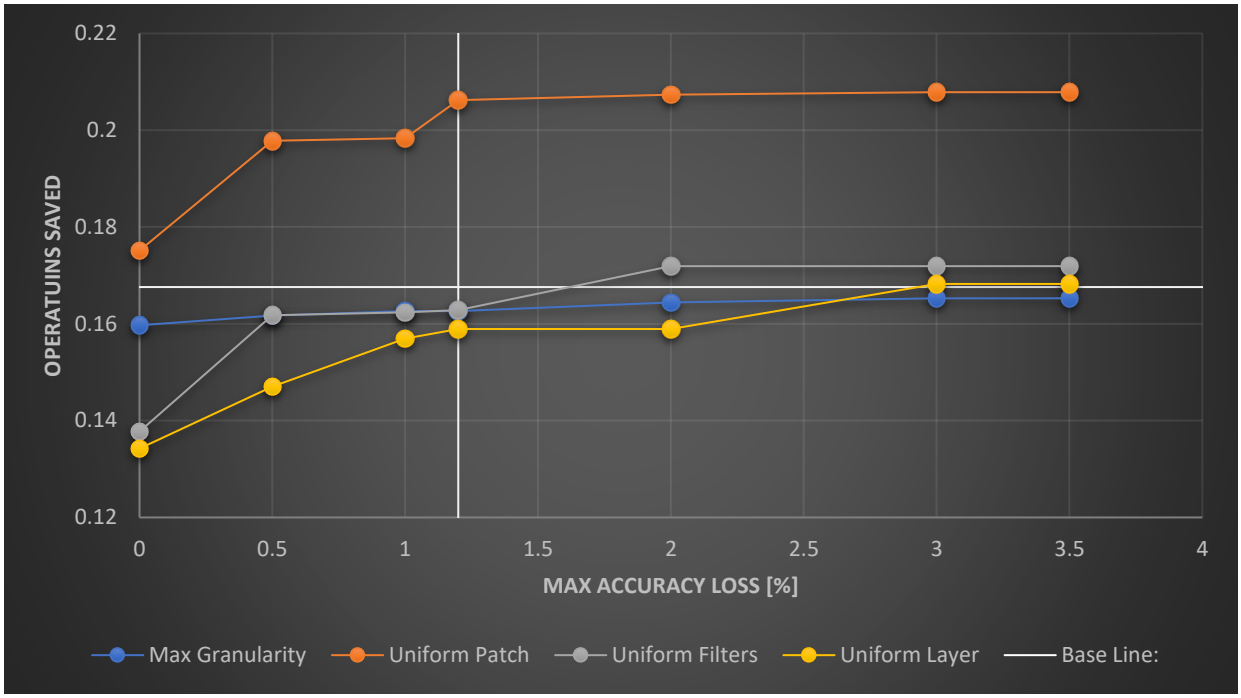

Figure 23: Operations saved for chosen values of maximal accuracy loss for Resnet18, MNIST, patch Size = 2, and Ones Range = [1,2]

6.3 AlexNet, CIFAR10, Five Spatial Layers

Network was trained to a 93.5% 1-0 classification accuracy on CIFAR10 (10 classes). We test our various optimization modes in contrast to the baseline result achieved in [1] – a uniform network diagonal mask. We receive a 14.1% decrease in accuracy, with 34.02% MAC operations saved for this configuration. GT parameter is set to a constant 10.

6.3.1 P=3, Ones Range of [1, 3] inclusive

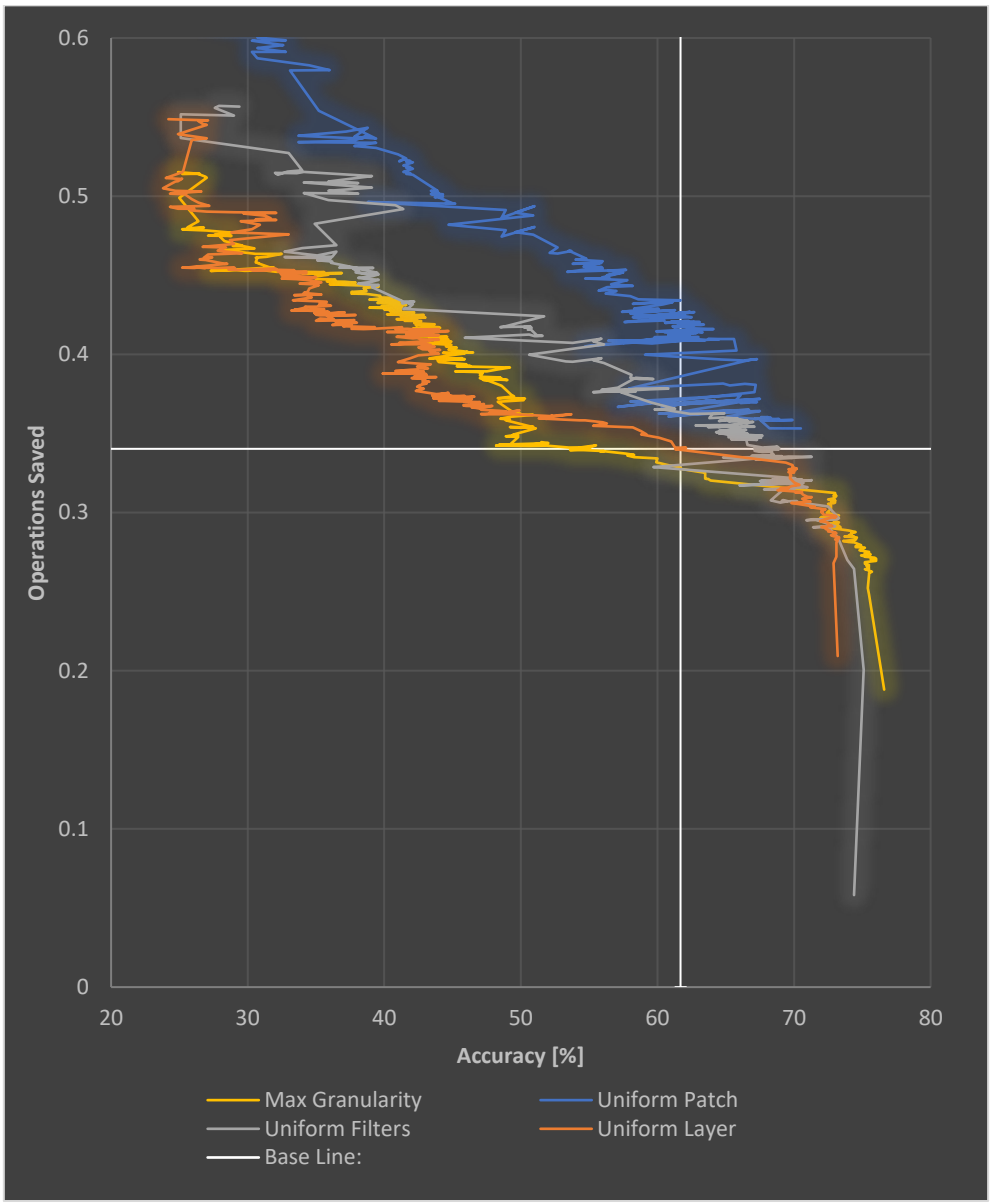

Figure 24: mode breakdown for AlexNet, CIFAR10, Patch Size = 3 and Ones Range = [1, 3]. The figure shows accuracy vs. operations saved for all the options generated in last stage of the optimization (the layer quantizer stage)

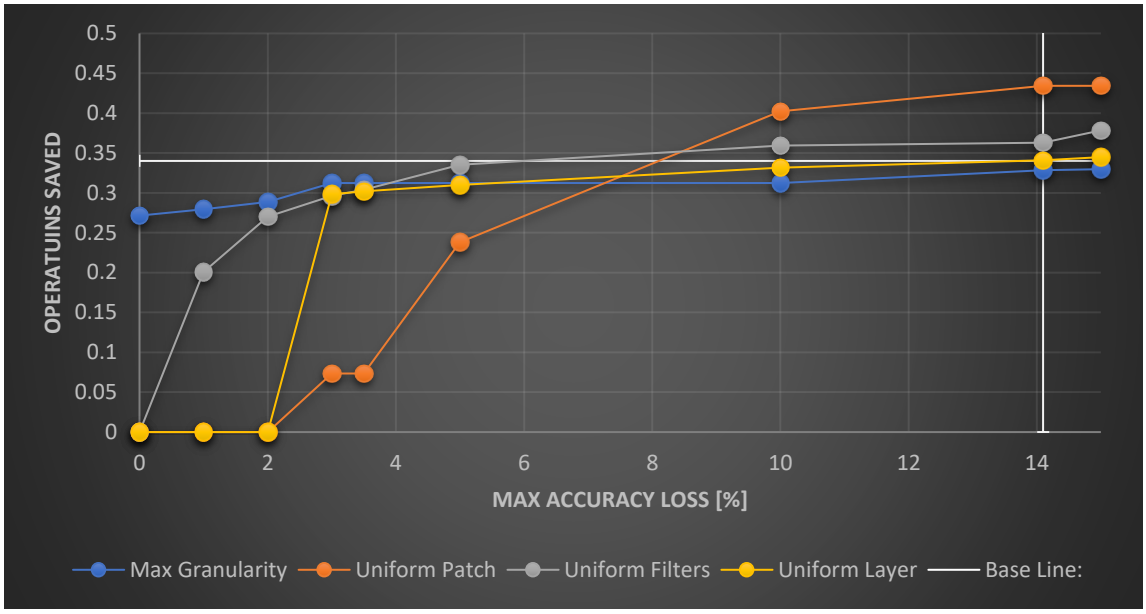

Figure 25: Operations saved for chosen values of maximal accuracy loss for AlexNet, CIFAR10, patch Size = 3, and Ones Range = [1,3]

6.3.1.1 Retraining Results

We performed a retraining of the AlexNet and CIFAR10 module with the final predictions masks we received after running the optimization algorithm. For each mode, we retrained the network with the masks received for the maximal accuracy loss showed in Figure 25. The retraining was done for 50 epochs with a learning rate of 0.01

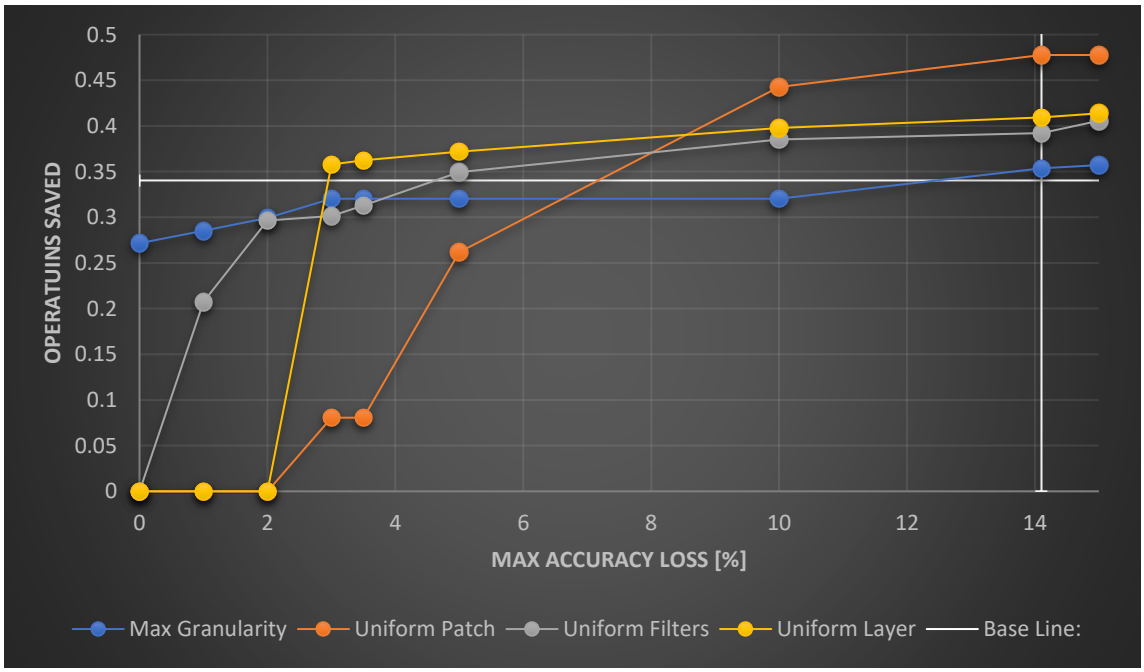

Figure 26: Operations saved for chosen values of maximal accuracy loss after retraining for AlexNet, CIFAR10, Patch Size = 3, and Ones Range = [1,3]. There are slight increases in the operations saved as compared to Figure 25

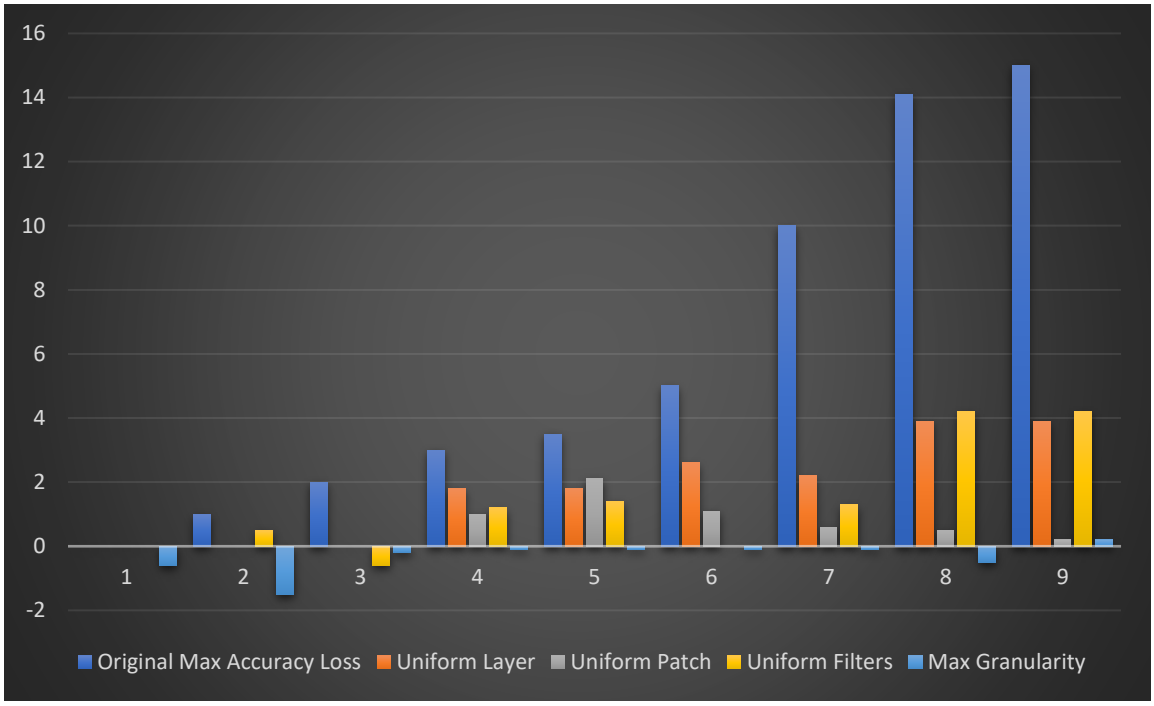

Figure 27: Accuracy loss before and after retraining

6.3.2 P=2, Ones Range of [1, 2] inclusive

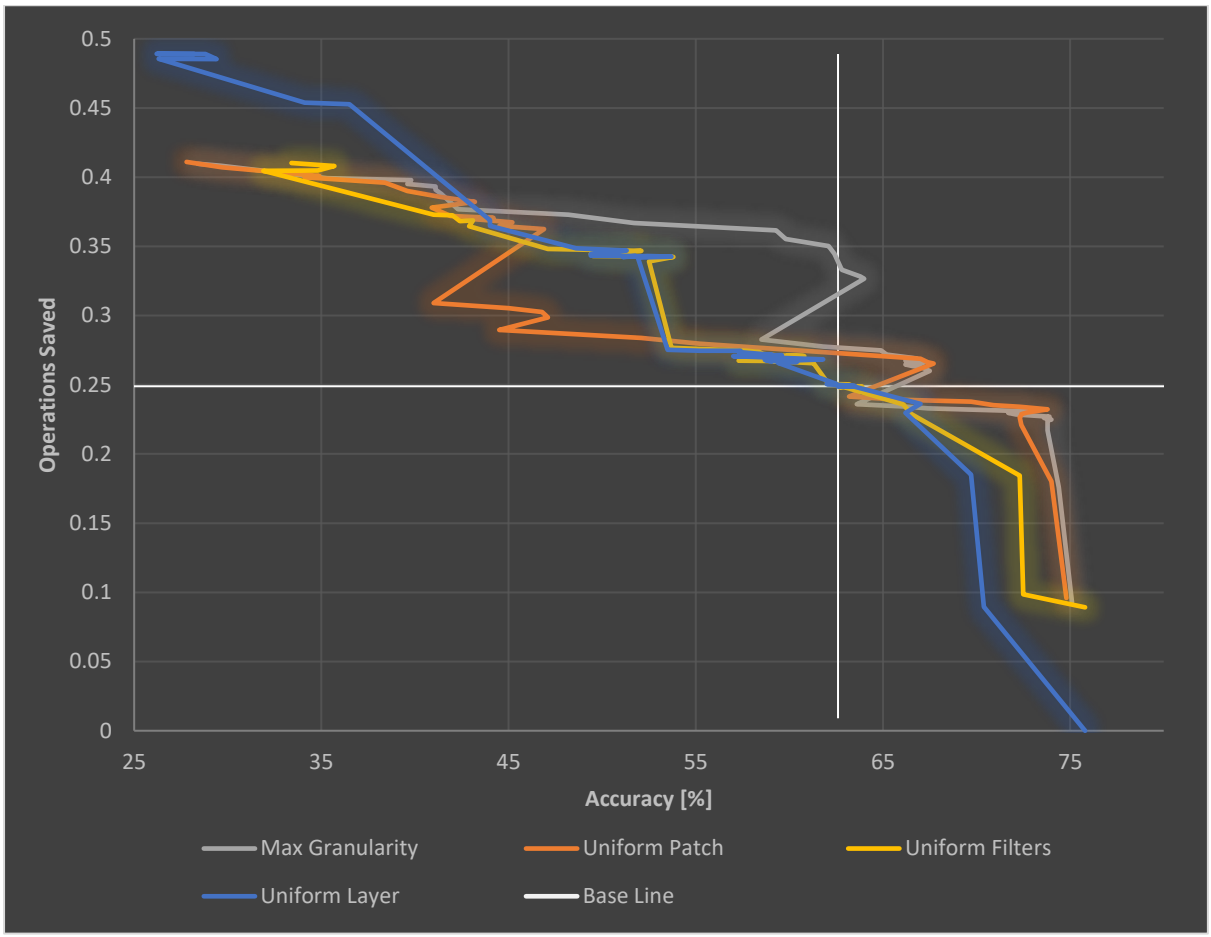

Figure 28: mode breakdown for AlexNet, CIFAR10, Patch Size = 2 and Ones Range = [1, 2]. The figure shows accuracy vs. operations saved for all the options generated in last stage of the optimization (the layer quantizer stage)

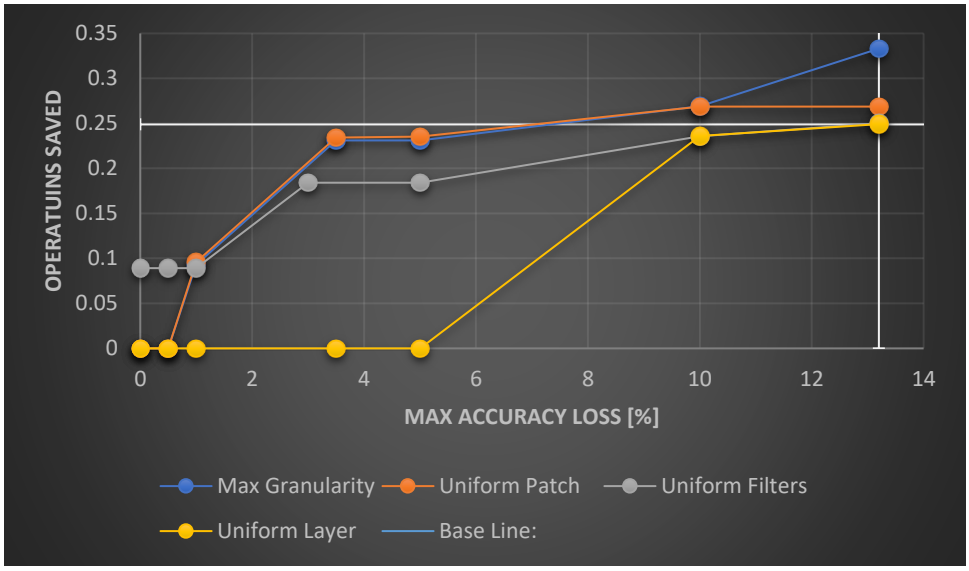

Figure 29: Operations saved for chosen values of maximal accuracy loss for AlexNet, CIFAR10, patch Size = 2, and Ones Range = [1,2]

7. Analysis

We compare between the project results, and the baseline results received with the diagonal predication mask described in

7.1 Top Results

Network & Dataset	AlexNet,CIFAR10		ResNet18, CIFAR10	ResNet18, MNIST
Prediction Window	3x3	2x2	2x2	2x2
MAC Reduction $\varepsilon = 0$	**27.15%**	**8.91%**	**13.69%**	**17.51%**
MAC Reduction $\varepsilon = 3$	31.24%	23.4%	25.32%	20.78%
MAC Reduction $\varepsilon = 5$	33.5%	23.53%	27.12%	21.01%
Baseline MAC Reduction	**34.02%**	**24.9%**	**23.5%**	**16.76%**
Baseline Accuracy Drop [%]	14.1	13.2	3.5	1.2
Our MAC Reduction for baseline accuracy	**42.42%**	**33.3%**	**25.32%**	**20.61%**

Table 1: Top results over all optimization modes and under different network, dataset and patch size setups

We note a substantial increase in MAC operation savings from the baseline results. Some networks and datasets allow for a more substantial optimization than others.

7.2 Top Retraining Results:

Results after retraining – AlexNet, CIFAR10, Prediction Window 3x3, Initial Accuracy 75.8
Baseline MAC Reduction: 34.02%, Accuracy Drop: 14.1

	Before Retraining	After Retraining	Increase in Accuracy [%]	Final Accuracy [%]
MAC Reduction $\varepsilon = 0$	27.15%	**27.16%**	+0.6	**76.4** (Higher than initial)
MAC Reduction $\varepsilon = 3$	31.24%	**35.79%**	+0.8	74
MAC Reduction $\varepsilon = 5$	33.5%	**37.2%**	+1.4	73.2
Our MAC Reduction for baseline accuracy	42.42%	**47.77%**	**+13.2**	75.3

Table 2: Top results after retraining for 20 epochs

We note a vast increase in accuracy under retraining (almost returning to the very same initial accuracy of the network) while increasing the number of MAC operations saved. With this retraining, we achieve a close to **50%** decrease in MAC operations with a mere 0.5% drop in accuracy.

8. Further Work

We feel that there are many more aspects to test with this framework. Possible ideas:

1. **Change of dataset** – We have yet to test the framework on many popular datasets, such as STL10 and ImageNet. The computational costs were too great to allow these more substantial datasets. How does the input size relate to our optimization powers?
2. **Testing of larger patch sizes** – We have shown promising results on AlexNet with a 3x3 patch. Do larger patches degrade the operation saving? For which value of patch size does the breakpoint exist?
3. **Effects of Retraining** - We show promising results after retraining, allowing us to almost completely regain the accuracy loss, while even increasing the operation savings. How far can we go with this? What is the best decrease in operation savings that is achievable for a given setup?
4. To be filled with more ideas

9. Bibliography

[1] G. Shomron and U. Weiser, "Exploiting Spatial Correlation in Convolutional Neural".

[2] Medium, "CNN Overview," [Online]. Available: https://medium.com/@udemeudofia01/basic-overview-of-convolutional-neural-network-cnn-4fcc7dbb4f17.

[3] S. CS231, "Training Neural Networks - Part 1," [Online]. Available: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture6.pdf.

[4] A. W. Harley, "An Interactive Node-Link Visualization," [Online]. Available: http://scs.ryerson.ca/~aharley/vis/conv/flat.html.

[5] V. Akhlaghi and A. Yazdanbakhsh, "SnaPEA: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks".

[6] S. Nayak, "LearnopenCV," [Online]. Available: https://www.learnopencv.com/understanding-alexnet/.

[7] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2015.

[8] A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks".

10. Appendix 1: Optimization Algorithm Pseudo Code

10.1 Inputs and Outputs

```
Inputs:     CNN      – a CNN model
            𝒟        – an optimization dataset
            ϵ        – maximal acceptable loss in classification accuracy
            mode     – mode of optimization, i.e. max granularity, uniform layer, uniform patch or
                       uniform filters
            ps       – patch size
            patterns– all possible patterns. Patterns can be selected directly or by specifying a
                       range of number of '1's in the patch
            GT       – granularity
outputs:    mask     – final mask
            LQrec    – LayerQuantizer stage results
            CQrec    – ChannelQuantizer stage results
            PQrec    – PatchQuantizer stage results
            FLrec    – first level results
```

10.2 Main Flow

```
initialize Optimizer(CNN, 𝒟, ϵ, mode, ps, patterns, gran)
FLrec = Optimizer.FirstLevelOptimizationPass()
PQrec = Optimizer. PatchQuantizerPass(FLrec)
CQrec = Optimizer. ChannelQuantizerPass(PQrec)
LQrec = Optimizer. LayerQuantizerPass(CQrec)
mask = FindFinalMask(LQrec, ϵ)
```

10.3 FirstLevelOptimizationPass

```
function FirstLevelOptimizationPass(Optimizer)
    initialize FLrec[layer][channel][patch][pattern] → None
    for layer ℓ in CNN do
        for channel c in ℓ do
            for patch 𝒑 in c do
                for patt in patterns do
                    tmp_mask = CreateFirstLvlMask(mode, ℓ, c, 𝒑, patt, GT)
                    ops_saved, err = Simulate(CNN, 𝒟, tmp_mask)
                    FLrec[ℓ][ c][𝒑][patt] = (ops_saved, err)
    return FLrec
```

10.4 PatchQuantizerPass

```
function PatchQuantizerPass(Optimizer , FLrec)
    for record in FLrec do
        if err > ϵ then
            remove record
    sort FLrec based on ops_saved
    initialize PQrec[layer][channel][pattern] → None
    for layer ℓ in CNN do
        for channel c in ℓ do
            for i in range(0, #patterns) do
                channel_pattern = CreateChannelPattern(mode, ℓ, c, FLrec[ℓ][ c][: ][i])
                ops_saved, err = Simulate(CNN, 𝒟, channel_pattern)
                PQrec [ℓ][ c][channel_pattern] = (ops_saved, err)
    return PQrec
```

10.5 ChannelQuantizerPass

```
function ChannelQuantizerPass(Optimizer , PQrec)
    for record in PQrec do
        if err > ϵ then
            remove record
    sort PQrec based on ops_saved
    initialize CQrec[layer][pattern] → None
    for layer ℓ in CNN do
        for i in range(0, #patterns) do
            layer_pattern = CreateLayerPattern(mode, ℓ, CQrec[ℓ][: ][i])
            ops_saved, err = Simulate(CNN, 𝒟, layer_pattern)
            CQrec [ℓ][layer_pattern] = (ops_saved, err)
    return CQrec
```

10.6 LayerQuantizerPass

```
function LayerQuantizerPass(Optimizer, CQrec)
    for record in CQrec do
        if err > ϵ then
            remove record
    sort CQrec based on ops_saved
    initialize LQrec[pattern] → None
    mask = CreateNetworkPattern(mode, CQrec[:][0])
    while mask is not None do
        ops_saved, err = Simulate(CNN, 𝒟, mask)
        LQrec[mask] = (ops_saved, err)
        mask = UpdateMask(mask, CQrec)
    return LQrec
```

10.7 UpdateMask

```
function UpdateMask(mask, CQrec)
        for layer ℓ in CNN do
                if CQrec[ℓ][curr_pattern+1] is not default pattern then
                                                // default pattern, i.e. ops_saved == 0
                        acc_diff =  CQrec[ℓ][curr_pattern+1].err - CQrec[ℓ][curr_pattern+1].err
                        curr_ops_saved = Sum(CQrec[:][curr_pattern].ops_saved)
                        next_ops_saved = curr_ops_saved - CQrec[ℓ][curr_pattern].ops_saved
                                                        + CQrec[ℓ][curr_pattern+1].ops_saved
                        ops_diff = (curr_ops_saved - next_ops_saved + 1)/( curr_ops_saved + 1)
                        merit[ℓ] = acc_diff/ops_diff
        if merit is not None then
                update layer in mask according to Max(merit)
                return mask
        else
                if all layers are set to default pattern then
                        return None
                else
                        update last layer in mask to default pattern
                        return mask
```

11. Appendix 2: Run Time Evaluation Example

Run time evaluation output for AlexNet network with CIFAR10 dataset, patch size 3x3:

```
=================================================================
-----------------------------------------------------------------
                      NET: AlexNetS
                  DATASET: CIFAR10
               PATCH SIZE: 3
                     ONES: 1-3
             GRANULARITY: 10
           TEST SET SIZE: 1000
    CHANNELQ UPDATE RATIO: 1
      PATCHQ UPDATE RATIO: 1
-----------------------------------------------------------------
    Mode.MAX_GRANULARITY                          1.59 [days]
-----------------------------------------------------------------
         iters 1st lvl         iters 2nd lvl          iters lQ
number          288960                150415               661
time             90156                 46929               196

sec per iter
        1st/2nd lvl: 0.312
        lQ: 0.296
-----------------------------------------------------------------
    Mode.UNIFORM_FILTERS                           0.3 [hours]
-----------------------------------------------------------------
         iters 1st lvl         iters 2nd lvl          iters lQ
number            2064                   650               656
time               685                   216               194

sec per iter
        1st/2nd lvl: 0.332
        lQ: 0.296
-----------------------------------------------------------------
    Mode.UNIFORM_PATCH                          13.78 [hours]
-----------------------------------------------------------------
         iters 1st lvl         iters 2nd lvl          iters lQ
number          148608                   650               656
time             49189                   215               194

sec per iter
        1st/2nd lvl: 0.331
        lQ: 0.296
-----------------------------------------------------------------
    Mode.UNIFORM_LAYER                           0.11 [hours]
-----------------------------------------------------------------
         iters 1st lvl         iters 2nd lvl          iters lQ
number              645                     0               651
time                207                     0               193

sec per iter
        1st/2nd lvl: 0.321
```

```
        lQ: 0.296
-------------------------------------------------------------------
===================================================================
```