University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Krystyna Gajczyk, Jakub Pierewoj, Przemysław Przybyszewski, Adam Starak
Student number: 332118, 360641, 332493, 361021

Inference in neural networks using low-precision arithmetic

Bachelor thesis in COMPUTER SCIENCE

Supervised by:
Dr. Konrad Durnoga

June 28, 2017

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Abstract

This project was created during "Team programming project" classes in 2016/17 and was proposed by Intel. We analyse some properties of binarized neural networks – the neural networks using binary parameters instead of floating-point numbers commonly used in deep learning. We experimented using different network structures and binarization algorithms to check accuracy and find ways to improve it.

Keywords

binarized neural network, XNOR-Net

Subject Area (Socrates/Erasmus code)

11.3 Informatics, Computer Science

Categories and Subject Descriptors

Computing methodologies - Machine learning - Machine learning approaches - Neural Networks

Thesis title in Polish

Inferencja w sieciach neuronowych przy użyciu arytmetyki niskiej precyzji

Contents

Chapter 1

Introduction

1.1. What is deep learning and why is it interesting?

Deep learning is a branch of machine learning which tries to model high level abstractions in the data (like understanding words or finding objects on image) using a graph of simple elements - neurons, connected with specific activations and weights to others. It has recently gained a lot of interest from the industry, especially after recent successes in image or speech recognision.

The most important part of deep learning is training a net based on samples, which classification is already known. After each training step one can change weights and activation in model to make output of the network as close as possible to the expected value.

Currently deep neural networks get better results than state-of-the-art algorithms in many areas, such as computer vision and speech recognition. This breakthrough was possible, because computing power of modern computers is high enough to handle intensive workload required to train large artificial neural networks.

1.2. Why are binarized networks interesting?

Deep neural networks (DNN) usually use high precision (floating- point) numbers to represent weights and activations. Computations using that arithmetic are usually handled by graphic cards. On devices with low computing power, such as mobile phones, use of deep neural networks is very limited.

Researchers tried to address this problem in the number of papers in the recent years. A couple of main approaches have been investigated in order to reduce inefficient computation and memory usage in deep neural networks while maintaining the classification accuracy. Those approaches can be summarized as:

- Shallow networks – it has been proven that every neural network can be replaced with a corresponding neural network with a single (possibly large) hidden layer. One of the main problems with this approach is, that in order to achieve similar accuracy as the original neural network, the shallow networks need approximately the

same number of parameters (numbers of neurons and connection between them). Second problem is that empirical test have shown, that while it shows good results for relatively small datasets, it underperforms for larger datasets (such as ImageNet).

- Compression of pre-trained DNN – it is possible to prune a DNN at inference time by discarding the weights, that do not bring any information to the classification process. It has been also further proposed [10] to reduce the number of parameters and/or activations in order to increase acceleration, compression. This reduces the main blockers for using DNN on small, embedded devices, such as memory usage and energy consumption. Memory usage reduction can be achieved through quantizing and compressing weights with Huffman coding or hashing weights, such that the weights assigned to one hash bucket share the same parameter.

- Compact layers – this approach also reduces the memory usage and computation time. This approach replaces parts of the DNN structure with corresponding elements, which are smaller in size and bring nearly as much information. A few techniques have been examined, such as replacing a fully connected layer with global average pooling and replacing a convolution with a corresponding one requiring smaller amount of parameters.

- Quantizing parameters – this technique aims to replace the floating-point parameters of the neural network with the quantized values (through vector quantization methods), which require smaller number of bits of memory and need simpler arithmetic operations for computation. Number of DNN with quantization have been designed: 8-bit integer instead of 32-bit floating point activations, ternary weights and 3-bit activations instead of floating points, etc. It was shown, that this approach can lead to a DNN representation which accuracy is not very far off from the state-of-the-art results.

- Network binarization – this is the extreme case of the parameter quantization technique. Due to a new learning algorithm, Expected Back Propagation [9], which is based on inferring network with binary weights and neurons through a variational Bayesian approach. Techniques that follow this approach mainly use real-valued weights as a reference for their binarization. This idea was further extended to binarize both weights and activations which was implemented in networks such as BinaryNet and XNOR-Net.

Each DNN has its own tolerance of inference precision. There are numerous topologies of neural networks. Each topology indicates the number of layers and defines how the neurons are connected. The aim of this project is to study the structures and predict which one will behave the best in our environment. Dealing with 1-bit integer weights is going to strongly affect the complexity of computational algorithms and the size of data. The whole workflow is going to be much faster. 1-bit operations are way simpler than the 32-bit ones. Furthermore, the

size of the inputs will require less memory. That is the perfect solution for less efficient devices.

Unfortunately, these adjustments will negatively affect the quality of prediction. Thus, the increase of the depth of a chosen net should be also taken into consideration. The loss of data and weights precision can be alleviated by the increased number of feature maps, i.e., the number of inputs for the next level neurons. We wanted to estimate the ratio between the number of feature maps in basic DNNs and the number of feature maps in low arithmetic precision DNNs that maintain similar quality of the prediction.

To sum up, BDNNs have some great properties, which should be investigated much deeper.

1.3. Our contribution

This project was created during "Team programming project" classes in 2016/17 and was proposed by Intel. Main goal of this project is the analysis of the effect that neural network binarization has on the inference quality of inference depending on:

- the network topology (LeNet, ResNet, AlexNet),
- the level of binarization (weights or weights and inputs).

We have also tested if it is possible to achieve the same accuracy as in the standard neural networks which use floating point values, just by increasing the number of filters in convolutional layers.

In addition to the main goal of this project, we have also added a binary convolution operation to TensorFlow. We hope it will be added to official TensorFlow distribution - we prepared a pull request with our implementation.

Chapter 2

Dictionary

For reader's convenience, we have decided to include a dictionary containing some of the terms that we use in this paper.

- Neural network – a machine learning model, inspired by the structure of a human brain. It consists of a large collection of neurons grouped into layers. Each neuron is connected to a set of other neurons. Each connection has an assigned value, also known as a weight, which measures its importance.
- Inference (forward propagation) – a process of computing neural network's output based on its input and values of neuron weights.
- Backward propagation – a process of updating weights based on the difference between the neural network output and the desired output.
- Activation – an output of a single neuron.
- Training – performing inference and backward propagation repetitively until a given stop condition (such as accuracy, the number of iterations or lack of accuracy improvement) is met.
- Convolution – a mathematical function defined as follows:

  $$(f * g)(t) = \int_{-\infty}^{\infty} f(t)g(t-\tau)d\tau$$

  where

  $$f, g \colon \mathbb{R} \to \mathbb{R}$$

- Convolution (discrete) – a mathematical function defined as follows:

  $$(f * g)(n) = \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

  where

  $$f, g \colon \mathbb{Z} \to \mathbb{Z}$$

- Convolutional layer – a neural network layer which performs a discrete convolution operation between its inputs and weights. The use of this kind of layers proved to be successful in the field of

image recognition, video recognition, and natural language processing.

- filters (or kernels) – weigths of convolutional layer.
- Mini batch – a small subset of input data. It is usually a good practice to perform weight update after processing a mini batch instead of a single element of the training set.
- Batch normalization layer – a layer which returns a modification of its input such that it has zero mean and unit variance.
- ReLU – a neuron activation function defined as follows:

$$R(x) = \max(0, x)$$

.

- Epoch – a single pass over all training set.
- Overfitting – an effect which occurs when machine learning model fits to the noise of the training sample instead of the underlying trend behind the data. It has a negative effect on the neural network performance on the real data.
- Regularization – a set of techniques which help to reduce overfitting.
- Dropout layer – a layer which returns a modification of its input such that some of the values are randomly set to zero. Dropout layers are used as a regularization technique. It takes parameter p which represents probability of kepping each parameter unchanged.

Part I

Architecture overview

Chapter 3

Binarization algorithms

Our approaches to neural network binarization are heavily inspired by the XNOR-Net paper. We have implemented two kinds of binarization: in the first one we only binarize weights, while in the second one we binarize both weights and activations. By binarization we mean mapping values from floating points into a set $-1, 1$.

3.1. Binarized weights

In this approach, binarized convolution layer output is computed as follows:

For each filter in this layer:

1. Let n be the total number of weights in filter, e.g., if a filter is a matrix 3×3 then $n = 9$.

2. Lest I be the layer's input.

3. Compute $W' = sign(W)$, where $sign$ is computed element-wise

4. Compute scalar value $\alpha = \frac{\sum |W_{ij}|}{n}$

5. Output $O = (W' \star I) * \alpha$, where $\star$ is a convolutaion operator. Convolution between W' and I can be camputed using only subtraction and addition, because W' contains only $1's$ and -1's.

It is proved in the XNOR-Net paper, that α minimizes the square mean error between O and $W \star I$.

The back propagation is the standard backward propagation made by using the original W matrix. To achieve this result, we override the standard gradient to change the sign to identity.

In this type of binarizations, the gradients are in full precision, therefore the backward-pass still requires convolution between 1-bit numbers and 32-bit floating-points.

3.2. Binarized filters and activations

For an original filter W and input I the forward propagation proceeds as follows:

1. Let n be the number of elements in each filter.
2. Compute $W_{sign} = sign(W)$.
3. Compute scalar value $\alpha = \frac{\sum |W_{ij}|}{n}$
4. Compute I_{abs} be a matrix with the absolute values of the input.
5. Compute $I_{sign} = sign(I)$ be a matrix of the signs of the input.
6. Let K be the result of the standard convolution using as input I_{abs} and as weights matrix k which has the same size as the input and is defined as follows: $k_{ij} = \frac{1}{n}$
7. Return $O = (I_{sign} \star W_{sign}) * \alpha * K$. Multiplying by K is element-wise.

It is worth noting that computing convolution in the last step can be done really efficiently. Both of operands contain only binary values, so it is possible to use only XNOR bitcount operations. Computation time savings are discussed in the next chapter.

On the other hand, binarization leads to a loss of information, so intuitively it should cause worse neural network performance. Effects of binarization on accuracy are our main contribution and are described in detail in next chapters.

Chapter 4

Advantages of binarized neural network

One of the advantages of using binarized neural networks is that we can save trained weights as binary numbers and store them on a device with limited resources. What is more, it is also possible to optimize the computation time. In this section, it is discussed exactly how much space and time we can save when using different kinds of binarization.

4.1. Binarized weights

In this case we store a binary value instead of a floating-point one for each weight. In addition, we also need to store coefficients α which are floating-point numbers. The amount of these coefficients depends on the type of binarization we perform. Computing α for each kernel compared to computing it for each layer gives us slightly worse space and computation time savings, but yields better accuracy.

A general formula for space savings is as follows (assuming we use 32-bit floating-point numbers):

$$s_m = \frac{space_{bin}}{space_{float}} = \frac{\#_\alpha * 32bit + \#_{weights} * 1bit}{\#_{weights} * 32bit}.$$

In case of different α for each layer $\#_\alpha = \#_{layers}$. So in a LeNet network, where $\#_{layers} = 4$ and $\#_{weights} = 1642272$ space saving coefficient $s_m = 1/(31.9975)$. If α is computed for each kernel then $\#_\alpha = \#_{kernels}$. To compute the space savings for the LeNet model, we can observe that the number of weights is the same as above but $\#_\alpha = 32 + 32 + 1024 + 10 = 1098$, so $s_m = 1/(31.3297)$.

If weights are binarized, every multiplication can be replaced with addition. This results in roughly two times shorter computation time [2].

4.2. Binarized inputs and weights (XNOR-Net)

Using this kind of binarization gives the same savings as for binarized weights, as we have to compute all β coefficients during inference (they

are not saved). Still, the space gain is around 32x which is an excellent result.

In this case, it is possible to compute convolution using XNOR bit-count operations only. This results in around 58 times better computation time [2].

Chapter 5

Networks and datasets

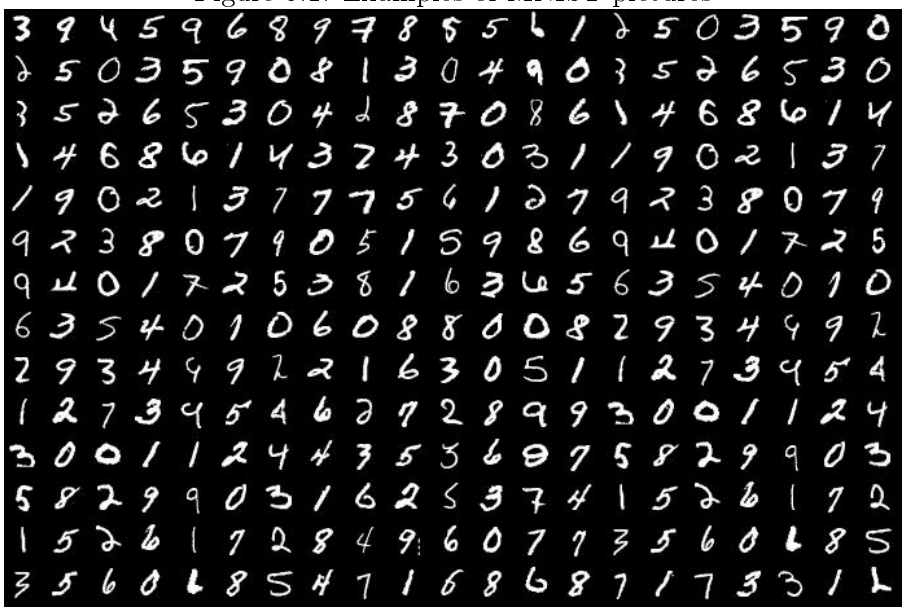
5.1. LeNet on MNIST

LeNet is a small 7-level network invented by Yann LeCun et al [7]. It is one of the best known neural network due to its simplicity. However, it does not mean that LeNet is not a powerful tool. According to Wikipedia, couple of banks apply LeNet to recognise written digits on cheques. The provided implementation was highly influenced by a Tensorflow's tutorial.

5.1.1. MNIST

MNIST is a set of handwritten digits drawn in black&white. Examples were resized and cropped so that each digit is centered and the size is equal to 18×18 pixels. The dataset is split into 2 parts: training (60,000 pictures) and test (10,000 pictures).

Figure 5.1: Examples of MNIST pictures

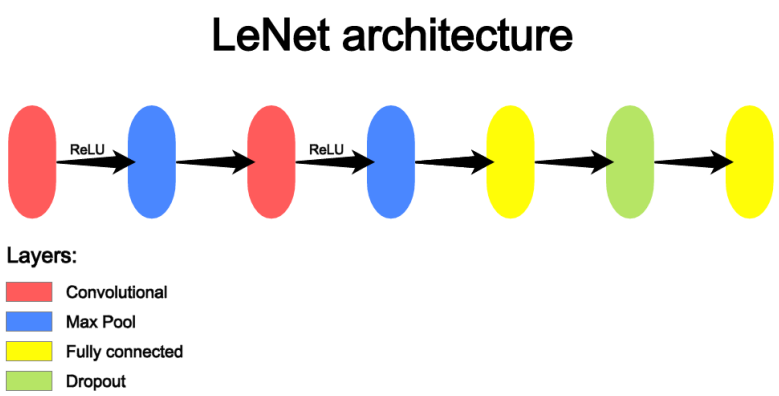


5.1.2. Network architecture

The general structure of network can be seen on figure 5.2. The exact structure is:

- Convolution layer: kernel size $5 \times 5 \times 32$
- Max Pool layer: kernel size 2×2
- Convolution layer: kernel size $5 \times 5 \times 64$
- Max Pool layer: kernel size 2×2
- Fully-connected layer: 1024 neurons
- Dropout layer with parameter 0.5
- Fully-connected layer: 10 neurons

Figure 5.2: Scheme of LeNet



5.1.3. Parameters

- Weight initialization: truncated normal, stddev 0.1
- Optimizer: Adam Optimizer, learning rate 10^{-4}
- Error: softmax cross entropy with logits
- Non-linearity: ReLU

5.2. AlexNet on ImageNet

AlexNet caused a breakthrough in deep neural networks after a great success during ImageNet Large-Scale Visual Recognition Challenge in 2012. It achieved top 5 test error rate of 15.4% leaving other opponents far behind. In comparison, the next team achieved an error rate of 26.2%.

5.3. Finetuned AlexNet on oxford-102

It has been 5 years since the astonishing victory of AlexNet, although people are still experimenting with it and achieve great results. jimgoo [8] proposed an ingenious approach of learning which gave him a top 1 error rate of 7% on oxford-102 set. In this experiment we try to replicate the results and find out whether it is possible to achieve a similar accuracy using binarized networks.

5.3.1. Oxford-flowers-102

The Oxford flowers dataset contains the most common flowers found in the United Kingdom. Each image represents a single flower. Within each class, the examples do not differ much but there exists a number of similar classes which makes the task harder. The images are saved in large resolution and their sizes are not normalized. The dataset is split into 3 parts: test (6,149 pictures), train (1,020 images) and validation (1,020 images). In this experiment, the machine is trained on the test set and tested against the train set as the authors of CaffeNet fine-tuned on the Oxford 102 category flower dataset did.

Figure 5.3: Examples of Oxford-flowers-102 pictures

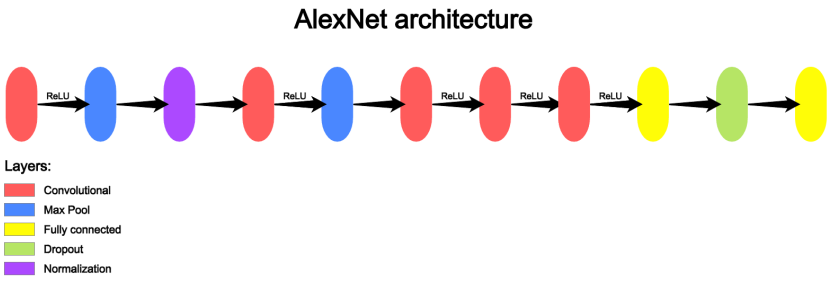
5.3.2. Network architecture

The general structure of network can be seen on figure 5.4. The exact structure is:

- Convolution layer: kernel size $11 \times 11 \times 96$, learning rate 10^{-3}
- Max Pool layer: kernel size 3×3

- Normalization layer
- Convolution layer: kernel size $5 \times 5 \times 256$, learning rate 10^{-3}
- Max Pool layer: kernel size 3×3
- Convolution layer: kernel size $3 \times 3 \times 384$, learning rate 10^{-3}
- Convolution layer: kernel size $3 \times 3 \times 384$, learning rate 10^{-3}
- Convolution layer: kernel size $3 \times 3 \times 256$, learning rate 10^{-3}
- Fully-connected layer: 4096 neurons, learning rate 10^{-3}
- Dropout layer: 0.5
- Fully-connected layer: 102 neurons, learning rate 10^{-2}

Figure 5.4: Scheme of AlexNet



5.3.3. Parameters

- Weight initialization: pretrained on ImageNet 2012 dataset
- Optimizer: stochastic gradient descent
- Error: softmax cross entropy with logits
- Non-linearity: ReLU

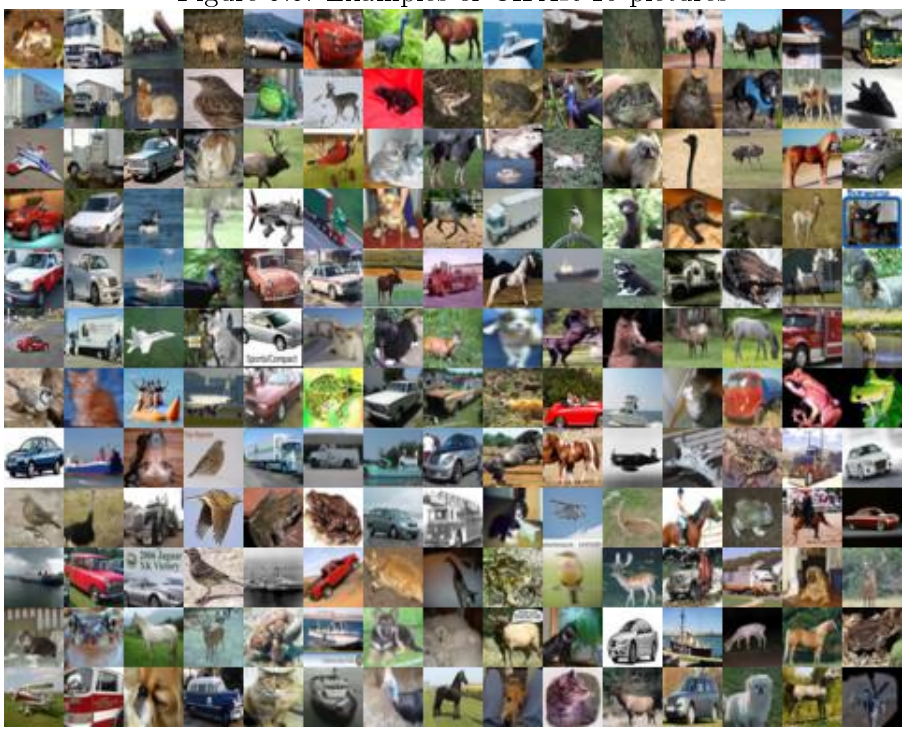
5.4. Residual network on CIFAR-10

Residual network, also called ResNet, was design in 2015 by Microsoft Researchers [4]. This network uses special identity edges to allow improvement on deeper networks.

5.4.1. CIFAR-10

CIFAR-10 is one of the most popular dataset. It is used for object recognition. It includes classes such as: airplanes, automobiles, birds, cats, deers, dogs, frogs, horses, ships, and trucks. This dataset contains 60,000 images in low resolution (32×32 pixels). Each class is represented by 6,000 images. The dataset was created by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton [6]. There also exists a bigger dataset called CIFAR-100.

Figure 5.5: Examples of CIFAR-10 pictures



5.4.2. Network architecture

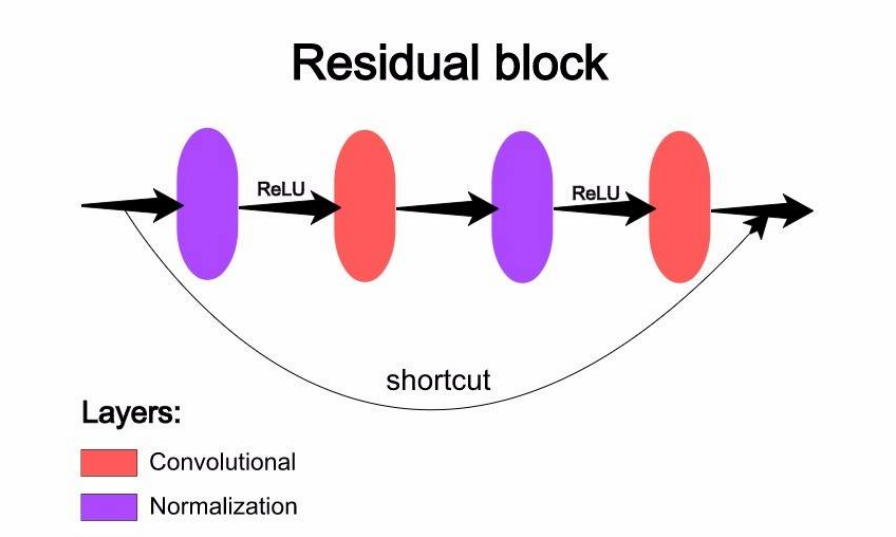
Resnet is constructed using blocks. Every block($num\_filters$, $size\_filters$) contains:

- Normalization layer
- Convolution layer with $num\_filters$ of $size\_filters$ size
- Normalization layer
- Convolution layer with $num\_filters$ of $size\_filters$ size
- Identity connection which adds input to output of second convolution

The general scheme of a single block can be seen on figure 5.6 We use blocks to construct Resnet(n) in a following way:

- convolutional layer of size matching input of network (for CIFAR it is 32)
- n blocks(16,32)
- pool
- n blocks(32,16)
- pool
- n blocks(64,8)
- normalization

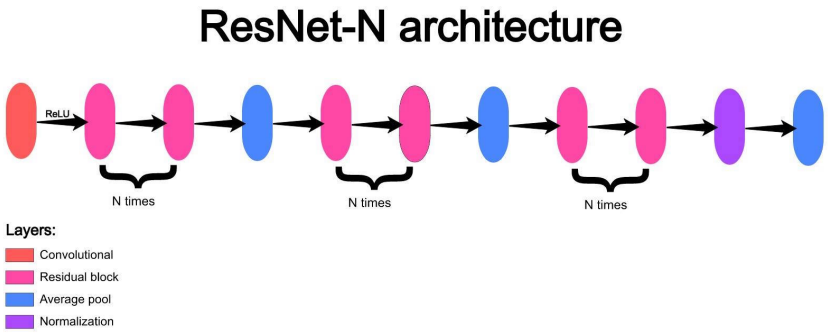
Figure 5.6: Scheme of single ResNet block



- pool

The general scheme of ResNet can be seen on figure 5.7.

Figure 5.7: Scheme of ResNet architecture



5.4.3. Parameters

- Weight initialization: XAVIER initializer
- Optimizer: momentum
- Error: softmax cross entropy with logits
- Non-linearity: ReLU

Chapter 6

The development process

6.1. Used software tools

A range of external tools was used in order to organise the work and to have a better overview of the current status of all of the ongoing tasks. Git was chosen as a version control system for this project mainly due to its advantages when it comes to local branching and that every member of the team had some experience with this tool.
Our next tool was Jupyter notebook, which enabled us to write and share code more easily and made us more productive.
When it comes to the build automation tool, there was no need to have one for the Python implementation and experiments. In case of the implemented C++ operation the team has used the build automation tool which is used by the Tensorflow framework, namely Bazel.
To simplify communication between members of the team and other people involved in this project we used Slack, where a list of dedicated channels was created. This has helped in ordering the discussions over different topics, as one topic was hold in one channel.

6.2. Used methodology

The team worked in accordance with the Scrum methodology with 1-week sprints. One of the members at the begining of the project was chosen to be the Scrum Master. Dividing workflow into separte tasks, each of which could have been accomplished in a week, helped keeping the team updated with the project status. Also, because of the weekly meeting every team member was updated about the work of the others. Due to the short length of the sprint, it was easier for the team to prioritize work and adapt to the changes in our approach of delivering the requested software.
Our Scrum board was created in a tool called Trello. In this tool one can define the stages that each task has to visit when moving from the backlog to done. It is also easy to assign work to specific people and to add subtasks for bigger tickets. During the whole project the backlog was getting updated after the weekly meetings. At the top of the backlog the tasks of the highest priority were kept.

6.3. Used machine learning framework

We decided to work with one of the open source frameworks, that can be used for neural networks. The structure of BNN proposed in Binary Connect [3] and XNOR-Net [2] is very similar to the standard convolutional network, so there is no need to implement BNN from scratch.

All frameworks that we analyse (TensorFlow, Torch, Caffe) have similar functionality, for example they have already implemented pooling and affine layers so we can easily reuse them.

From available frameworks we decided to choose TensorFlow [5]. It is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (called tensors) transported between them. In case on of neural networks, nodes will represent layers of network. The advantage of TensorFlow over other frameworks is really good community support and solid documentation.

To perform our experiments more efficiently, we have used a higher level API for TensorFlow: TFLearn, which is a great tool for fast prototyping and creating proofs of concepts. It also provides great features for neural network visualization, such as weights plotting.

Chapter 7

Implementation details

7.1. Implementation in Python

Tensorflow Python library offers numerous built-in functions for machine learning and data exploring. Every operation is supplied with modifiable gradient. It significantly lessens the amount of effort and code written by scientists. There are more advantages of using built-in tensorflow's functions. First of all, almost every Tensorflow's operation provides a GPU implementation. It is a crucial feature. The computing power of GPU devices is much greater than CPU's one. Also, GPUs have an ability to perform more computations in parallel, by which the models trained on GPU are computed several times quicker. Secondly, our binarized operations are less likely to have implementation bugs. Assuming that Tensorflow's functions compute correctly, the only thing to do is compose them and modify the gradients so that it will fit XNOR-Net authors' equations. It is a great advantage, especially on early stage, because future C++ operation can be based on it.

Tensorflow library provides a wrapper for user-defined functions. We used it at the beginning, but it quickly turned out that writing a custom function leads into trouble. The approach got changed. Instead of writing own function, we started to compose built-in ones. The most tricky part comes when parameters need to be binarized. On one hand, in feed-forward, the convolution must be computed with binary parameters. On the other hand, in backward propagation, the gradient must be computed with respect to the full-precision parameters. First part of this problem was solved by applying signum function. Unfortunately, this application affects the gradient computation. Changing the gradient function of signum to identity fixed the second part of the problem (this fix was firstly applied in DoReFa-Net). The implementation supports both weights and weights & inputs binarization. The second method does not compute redundant factors, i.e., the second technique described in XNOR-Net paper [2] is used.

7.2. Implementation in C++

In Tensorflow framework it is possible to add a so-called core user-defined operation to the existing ones. To support binary convolution

two operations were added, namely BinaryConv2D, which binarizes only filter weights, and BinaryConvInput2D binarizing both inputs and filters' weights. Those were implemented in source files `conv_bin.cc` and `conv_bin_inputs.cc`, respectively. The source files were added to `tensorflow/core/user_ops` directory.

The first part of both kernel implementations is validation of user provided data, e.g., checking whether filter's and input's depths are equal and whether the strides attributes are well-defined.

The second part is where the calculation of binarized convolution is conducted. Its design follows the algorithms described in Chapter "Implemented binarization algorithms". In both added operations a parameter 'alpha' is being calculated. The speed-up in its calculation was gained by using OpenMPI library and dividing the work of calculating summation of absolute values of filter's weights across all existing CPU cores.

In the BinaryConvInput2D kernel operation also calculation of parameter 'beta' is conducted. This required to calculate averaged inputs across channels and, for each output's cell, to calculate an appropriate 'beta' parameter. Those parameters are stored in a matrix 'K'. Also in this case some speed-ups were implemented. In a naive implementation, calculation of matrix 'K' performed repeated operations of calculating a sum over the same subset of input data. To avoid summing the same elements many times, dynamic programming approach was used, in which the incremental sums are stored in a designated matrix 'D'.

The last action is the binarized convolution. After binarizing the appropriate parts on data the resulting output of the convolution is multiplied by the corresponding 'alpha' and 'beta' parameters. Also in this case the operation was implemented in a way that its calculation could be parallelized using all available CPU cores.

Both implementations follow the typical scheme of a Tensorflow operation. They consist of two sections - in one an operation is declared (`REGISTER_OP`) and in the second one the registered operation is linked to the given kernel that works on specific types and on a specific device (`REGISTER_KERNEL_BUILDER`). Our both implementations work only on CPU and on float types.

A special build script was added to compile our source files into a shared object file which was then manually added in our Python tests and programs by a special Tensorflow call `load_op_library`.

In order to test if the kernels computations are correct, unit tests were added to the library. Those tests are run every time the shared object file is created.

Chapter 8

Team members contribution

The team worked together for 2 academic semesters. In sections below we listed individual contribution of each member of out team.

8.1. Krystyna's contribution

- organizing the team's work (Scrum Master)
- contact with project supervisor
- implementation of Resnet
- Resnet experiments

8.2. Jakub's contribution

- implementation of TensorFlow binarized convolution
- LeNet experiments
- implementing C++ gradient for new operation

8.3. Przemysław's contribution

- contact with our partner company
- implementation of TensorFlow binarized convolution
- implementing quality tests for project

8.4. Adam's contribution

- implementation of AlexNet
- experiments on AlexNet
- code review

Part II

Experiment results

Chapter 9

Binarized vs standard network accuracy

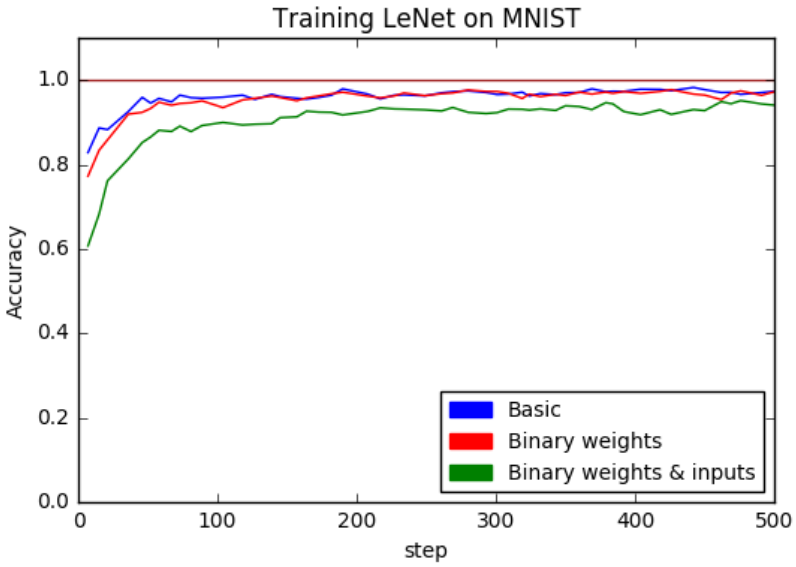
9.1. LeNet on MNIST

9.1.1. Results and discussion

Results The results of experiments are given in Table 9.1.

Table 9.1: LeNet accuracy on MNIST

Full-precision	Binary weights	Binary weights & inputs
98%	98%	95%



Discussion Both binarization methods acquired near state-of-the-art scores. The MNIST dataset is really simple and the LeNet structure is not complicated. Thus, the binarization does not significantly affect the precision. The results are neither surprising nor ground-breaking. We made this experiment mostly for verifying our binarization methods.

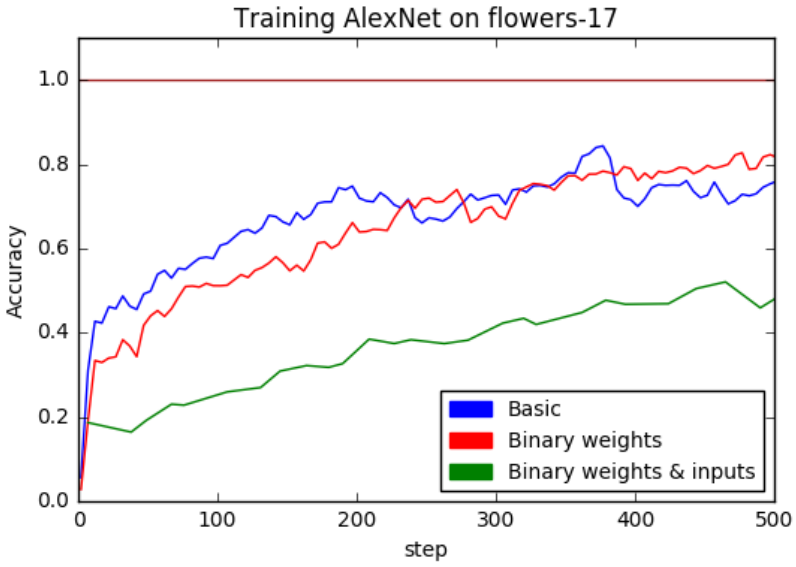
9.2. AlexNet on Flowers

9.2.1. Binarization results and discussion

The results of experiments are given in Table 9.2.

Table 9.2: AlexNet accuracy on flowers-17 (50 epochs)

Accuracy	Full-precision	Binary weights	Binary weights & inputs
top-1	76%	79%	64%
top-5	98%	95%	90%



Discussion The results are promising considering the input images have high resolution. This experiment shows that fully binarized network does not loose as much precision as it was expected. Surprisingly, the binary weights network outperforms the full-precision one.

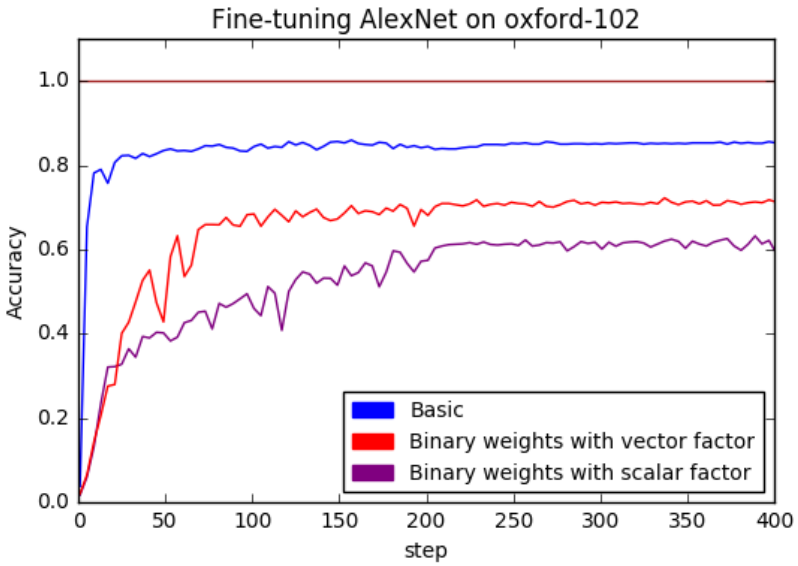
9.2.2. Fine-tuning results and discussion

The results of experiments are in table 9.3.

Table 9.3: Fine-tuning AlexNet on flowers-102

Full-precision	Binary weights vector factor	Binary weights scalar factor
85%	72%	63%

Discussion This experiment shows that binarized networks achieve near state-of-the-art performance also with pretrained weights, which were not trained on the binarized network.



9.3. Residual network on CIFAR-10

9.3.1. Results and discussion

Binary weights

Results The results of experiments are given in Table 9.4.

Table 9.4: Results of Resnet and BinResnet

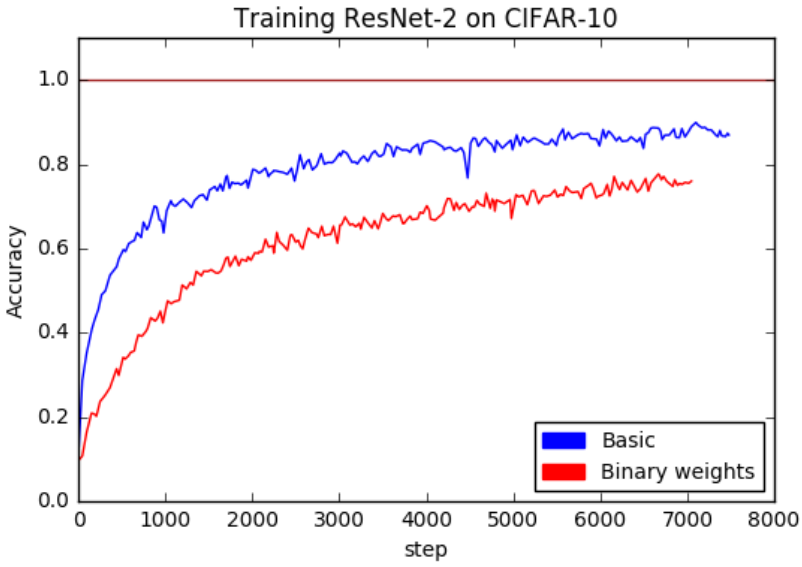
blocks	layers	learning rate	epochs	ResNet	BinResNet
2	14	0.1	15	80%	77%
5	32	0.1	15	82%	81%
18	110	0.1	20	84%	82%
2	14	0.01	15	78%	76%
5	32	0.01	15	79%	78%
18	110	0.01	20	81%	80%

Discussion The results of binarized network are very good comparing to standard ResNet. The property of ResNet is preserved and the difference in accuracy is not significant. It is an interesting result as it shows that binarizing ResNet only slightly decreases its accuracy. Taking this into consideration, it is worth to use binarization to save memory and computation time, especially on CPU.

Binary Resnet with binary weights and inputs

Discussion We did not manage to train ResNet of any size with binary weights and inputs, though the authors of XNOR-Net claim that their network achieves near state-of-the-art results. They did not provide any code, so we were struggling with it on our own. The size of ResNet makes it very delicate – a small change in the learning rate has a great impact on the accuracy. In this experiment, the best, yet still

Figure 9.1: Training ResNet-2 on CIFAR10



unsatisfactory, accuracy was achieved by ResNet-2 with the score of 20%. During experiments we strugled with following issues:

- An implementation which uses `tf.nn.conv2d` twice in each binarization process is much slower.
- A network with both weights and activations binarized is much harder to learn. In a 2-layer Lenet the results were still very stable but adding more layers, for instance 14, in a 2-blocks ResNet made network able to learn only during the first few rounds of computation.
- Smaller learning rate allows network to learn. Unfortunately, the problem of deeper network still occurs, so the advantage of ResNet is not preserved.

Chapter 10

Increasing the number of binarized filters

10.1. LeNet on MNIST

10.1.1. Discussion

Every binarization of LeNet achieved near state-of-the-art results. An experiment in which we increase the amount of feature maps in LeNet yielded non-interesting results because there results were already very good.

10.2. AlexNet on flowers-102

10.2.1. Discussion

Combining pretrained feature maps with randomized ones leads to troubles. No network managed to converge. Only if the randomized parameters are set to values near 0, the networks start to learn. The problem is that new parameters do not have significant meaning, so this experiment reduces to the one in the previous section. On the other hand, our resources are too small to obtain pretrained, scaled filters, because it requires to train AlexNet on the ImageNet dataset, which has over 1.2 million of images.

10.3. AlexNet on flowers-17

10.3.1. Discussion

Weights binarization led to the state-of-the-art performance, so we omit this case. Unfortunately, we did not manage to achieve greater performance on the binary weights & inputs network with increased number of feature maps. The more we increased the amount of feature maps, the worse accuracy on testing set we achieved (3% on average), though the training set accuracy rose (from 73% to 80%). It also requires about 3 times longer training. We were not able to run networks with higher feature map ratio than 2, because the size of the network exceed the resources of our laptops.

10.4. ResNet on CIFAR-10

10.4.1. Results

The results of experiments are given in plot 10.1.

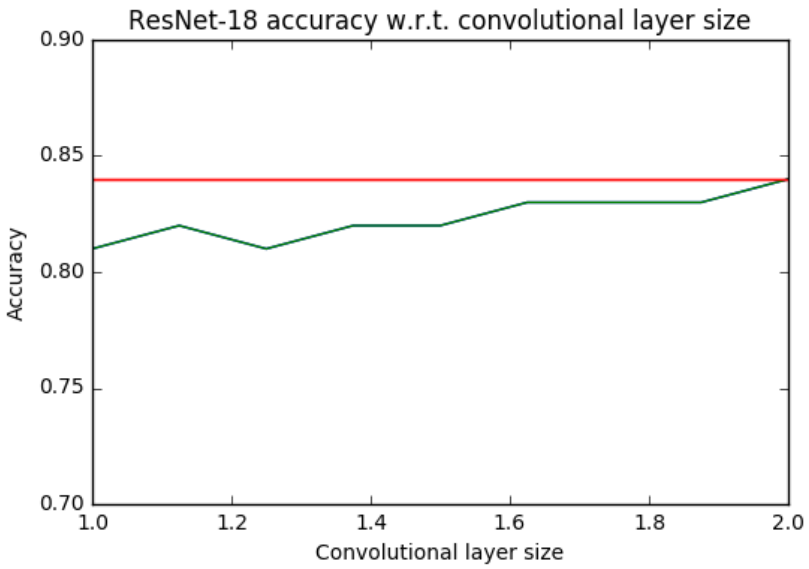


Figure 10.1:

10.4.2. Discussion

Increasing the number of filters improves network accuracy. If the number of filters in each convolutional layer is doubled we achieve the accuracy of a non-binarized network.

Chapter 11

Conclusions

11.1. Discussion of experiments results

Our experiments confirmed our thesis – binarized neural network can achieve accuracy which is very close to their respective non-binarized equivalents. It works pretty well with simple architectures such as LeNet and AlexNet. On more complicated network, such as ResNet, one must be very careful when setting up parameters.

For partially binarized network (with only filters binarized) we obtained great results for all tested architectures.

Increasing the number of filters by around 100% for each binarized network allowed us to achieve accuracy as good as the one of the original one. This result is very promising – if we take twice as many parameters but each of them is 32 times shorter (after converting a float to a boolean value) we still use 16 times less memory.

11.2. Future work

There are several aspects that could benefit from futher analysis, namely:

- Only three ANN structures were tested. Those experiments could be extended to include also other popular structures such as GoogLeNet, VGG Net and ZF Net.
- Kernels in the C++ implementation are designed only for CPU. For a potential speed-up gain new implementations of those operations on GPU could be added.
- The current C++ implementation for binarizing both inputs and filter's weights does not save any memory (it uses as much memory as the standard convolution operation does). This could be improved by adding support for one or two bit types.
- We have only checked to what extent the binarized network needed to be increased to achive the accuracy of its standard version. It would be interesting to check what is the limit of achievable accuracy for the binarized networks and how many iterations are required to achieve the maximal accuracy on a given dataset.

Bibliography

[1] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, Oriol Vinyals, *Understanding deep learning requires rethinking generalization*, https://arxiv.org/abs/1611.03530 (2016)

[2] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, Ali Farhadi, *XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks*, https://arxiv.org/abs/1603.05279 (2016)

[3] Matthieu Courbariaux, Yoshua Bengio, Jean-Pierre David, *BinaryConnect: Training Deep Neural Networks with binary weights during propagations*, https://arxiv.org/abs/1511.00363 (2015)

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, *Deep Residual Learning for Image Recognition*, https://arxiv.org/abs/1512.03385 (2015)

[5] https://www.tensorflow.org/

[6] https://www.cs.toronto.edu/~kriz/cifar.html

[7] http://yann.lecun.com/exdb/lenet/

[8] https://github.com/jimgoo

[9] Daniel Soudry, Itay Hubara, Ron Meir, Expectation Backpropagation: Parameter-Free Training of Multilayer Neural Networks with Continuous or Discrete Weights, https://papers.nips.cc/paper/5269-expectation-backpropagation-parameter-free-training-of-multilayer-neural-networks-with-continuous-or-discrete-weights.pdf

[10] Xing Wang, Jie Liang, Scalable Compression of Deep Neural Networks, https://arxiv.org/abs/1608.07365 (2016)