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 24, 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

TODO

Keywords

binarized neural network, XORNET

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

1. Introduction 5
1.1. What is deep learning and why is it interesting? 5
1.2. Why binarized networks are interesting? 5
1.3. Our contribution 7

2. Dictionary 9

3. Implemented binarization algorithms 11
3.1. Binarized convolution filters from XORNET [2] 11
3.2. Binarized convolution filters inspired by DoReFa [4] . . . 11
3.3. Binarized filters and activations 12

4. Advantages of binarized neural network 13
4.1. Binarized weights 13
4.1.1. Different α for each layer 13
4.1.2. Different α for each kernel 13
4.2. Binarized inputs and weights (XNORNET) 13

5. Used networks and datasets 15
5.1. LeNet on MNIST 15
5.1.1. MNIST 15
5.1.2. Network architecture 16
5.1.3. Parameters 16
5.2. AlexNet on ImageNet 16
5.3. Finetuned AlexNet on oxford-102 17
5.3.1. Oxford-flowers-102 17
5.3.2. Network architecture 17
5.3.3. Parameters 18
5.4. Residual network on CIFAR-10 18
5.4.1. CIFAR-10 18
5.4.2. Network architecture 19
5.4.3. Parameters 20

6. Implementation details 21
6.1. Used framework 21
6.2. Our implementation 21
6.2.1. Implementation in Python 21
6.2.2. Implementation in C++ 21

7. Experiments results 23
7.1. Lenet on MNIST 23
7.1.1. Results and discussion 23
7.2. AlexNet on Flowers 23
7.2.1. Binarization results and discussion 23
7.2.2. Fine-tuning results and discussion 24
7.3. Residual network on CIFAR-10 24
7.3.1. Results and discussion 24

8. Conclusions 27
8.1. Discussion of experiments results 27
8.2. Future work 27

9. Team members contribution 29
9.1. Krystyna's contribution 29
9.2. Jakub's contribution 29
9.3. Przemysław's contribution 29
9.4. Adam's contribution 29

Bibliography 31

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 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 net based on the samples that the result is 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 binarized networks are 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 proved, 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 don't bring any information to the classification process. It has been also further proposed 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. Memory usage 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, which bases on inferring network with binary weights and neurons through a variational Bayesian approach. Techniques using this approach mainly use the 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, those properties will negatively affect the quality of prediction. Thus, the increase of the depth of chosen net should be also taken into consideration. A feature map is an input for the next level neurons. The loss of data and weights precision can be alleviated by the increased number of the feature maps. We wanted to estimate ratio between number of feature maps in basic DNN and number of feature maps in low arithmetic precision DNN that maintain similar quality of the prediction. To sum up, the research may bear out, that BDNNs have some great properties, which should be investigated much deeper. It may also find out, that they achieve better results in some cases and the present solutions should be replaced with BDNN.

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:

- network topology (LENET, ResNet, AlexNet)
- level of binarization (weights or weights and inputs)

We have also tested if it is possible to achieve same accuracy as 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.

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 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 neural network output and the desired output.
- Activation - output of a single neuron.
- Training - repetitively performing inference and backward propagation until given stop condition (such as accuracy, 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

  $$f, g : R \to R$$

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

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

  where

  $$f, g : Z \to Z$$

- Convolutional layer - a neural network layer which performs a discrete convolution operation between its inputs and weights (in this case they are usually called weight filters or kernels). Usage

of this kind of layers proved to be successful in the field of image recognition, video recognition and natural language processing.

- 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 - one pass over all training set.
- Overfitting - 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 - 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.

Chapter 3

Implemented binarization algorithms

3.1. Binarized convolution filters from XORNET [2]

In this approach we use special binarized filter for forward propagation.

For an original filter W forward propagation is:

1. Let n be the number of elements in each filter, e.g. if filter is matrix $3 \times 3, n = 9$.
2. Let W' be matrix containing signs of W.
3. Calculate A - average of elements for each filter in W', so A is a vector of size equal to number of filters in layer.
4. Compute standard convolution using matrix W'.
5. Return result of convolution multiplied by A.

The back propagation is standard back-propagation made by using original W matrix. To achieve this result we override standard gradient to change 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 convolution filters inspired by DoReFa [4]

In this approach we do not implement full methods from DoReFa network. We worked on one idea to average filters over all maps at the same time. It is very similar to XORNET scheme.

TODO - opis dokładniejszy co analizuje DoReFa

For original filter W forward propagation is:

1. Let n be the number of elements in each filter, e.g. if filter is matrix $3 \times 3, n = 9$.
2. Let W' be matrix containing signs of W.
3. Calculate A average of elements for all filters in W, so A is a scalar.
4. Compute standard convolution using matrix W'.
5. Return result of convolution multiplied by A.

The only difference is that A is now scalar, not a vector. This approach allows to speed up computation for both forward and back propagation (multiplication by scalar is very efficient) comparing to XORNET.

3.3. Binarized filters and activations

For original filter W and input I forward propagation is:

1. Let n be the number of elements in each filter, e.g. if filter is matrix 3x3, n=9.
2. Let W_{sign} be matrix containing signs of W divided by A.
3. Calculate A average of elements for each filter in W_{sign}, so A is a vector of size equal to numbers of filters in layer.
4. Let I_{abs} be matrix with absolute value of input.
5. Let I_{sign} be matrix with signs of input.
6. Let K to be result of computation of standard convolution using as input I_{abs} and as weights matrix containing $\frac{1}{n}$ on each position.
7. Compute standard convolution using input I_{sign} and weights W_{sign}.
8. Return result multiplied by K and A.

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. In this section, it is discussed exactly how much space we can save, when using different kinds of binarization.

4.1. Binarized weights

In this case we store binary value instead of a floating point 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 foe each layer gives us slightly worse space and computation time savings, but yields better accuracy.

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

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

4.1.1. Different α for each layer

In this case $\#_\alpha = \#_{layers}$. So in in LENET, where $\#_{layers} = 4$ and $\#_{weights} = 1642272$ we have $s_m = 1/(31.9975)$.

4.1.2. Different α for each kernel

In this case we have $\#_\alpha = \#_{kernels}$. To compute the space savings for LENET model, we can observe that the number of weights is same as above, but $\#_\alpha = 32 + 32 + 1024 + 10 = 1098$, so $s_m = 1/(31.3297)$.

4.2. Binarized inputs and weights (XNORNET)

Using this kind of binarization gives same savings as 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.

Chapter 5

Used networks and datasets

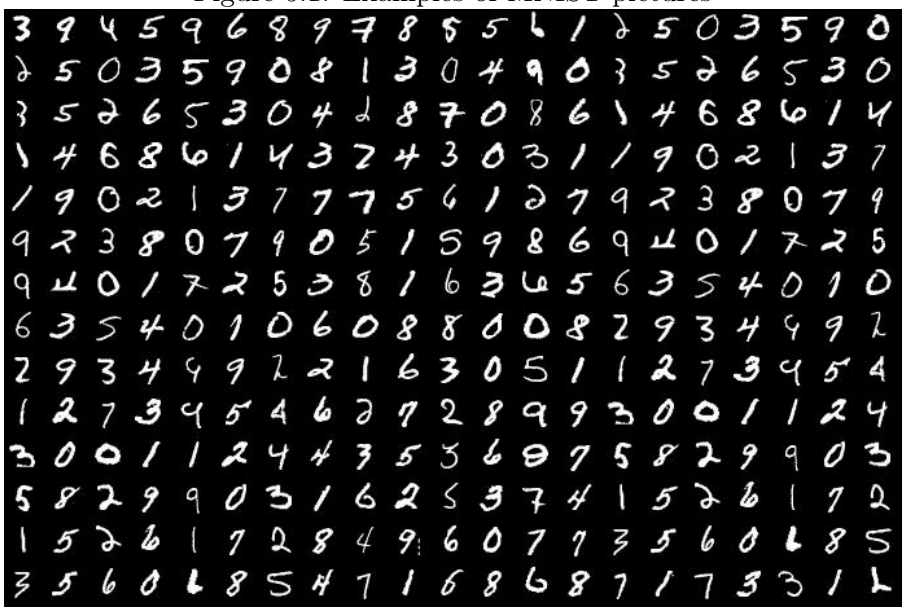
5.1. LeNet on MNIST

LeNet is a small 7-level network invented by Yann LeCun et al. It is one of the most known neural network due to it's simplicity. Almost everyone, who is new to machine learning, begins with implementing it on MNIST dataset. However, it does not mean that LeNet is not a powerful tool. According to Wikipedia, couple of banks are applying LeNet to recognise written digits on cheques. Provided implementation was highly affected by Tensorflow's tutorial.

5.1.1. MNIST

MNIST is a subset of NIST. Each example represents a handwritten digit 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).

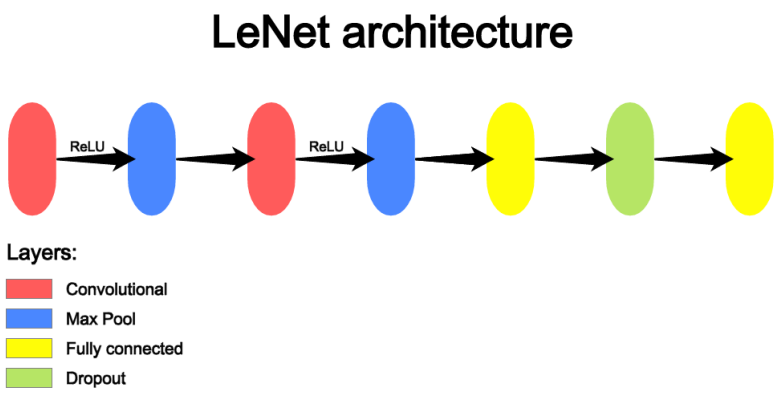
Figure 5.1: Examples of MNIST pictures



5.1.2. Network architecture

- 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: 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 huge 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 achieving great results. jimgoo(nick z gita, trzeba bedzie zacytowac) proposed a very clever 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 similar accuracy using binarized network.

5.3.1. Oxford-flowers-102

Oxford flowers dataset is containing the most common flowers found in United Kingdom. Each image represents a single flower. Within each class, the examples do not differ much, but there exists many similar classes, which makes the task harder. The images are saved in large scale and its' 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.

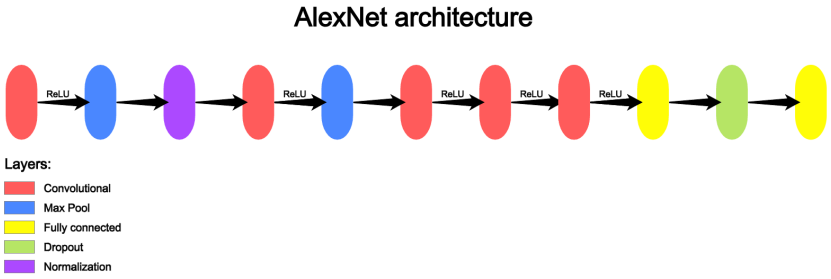
Figure 5.3: Examples of Oxford-flowers-102 pictures

5.3.2. Network architecture

- 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

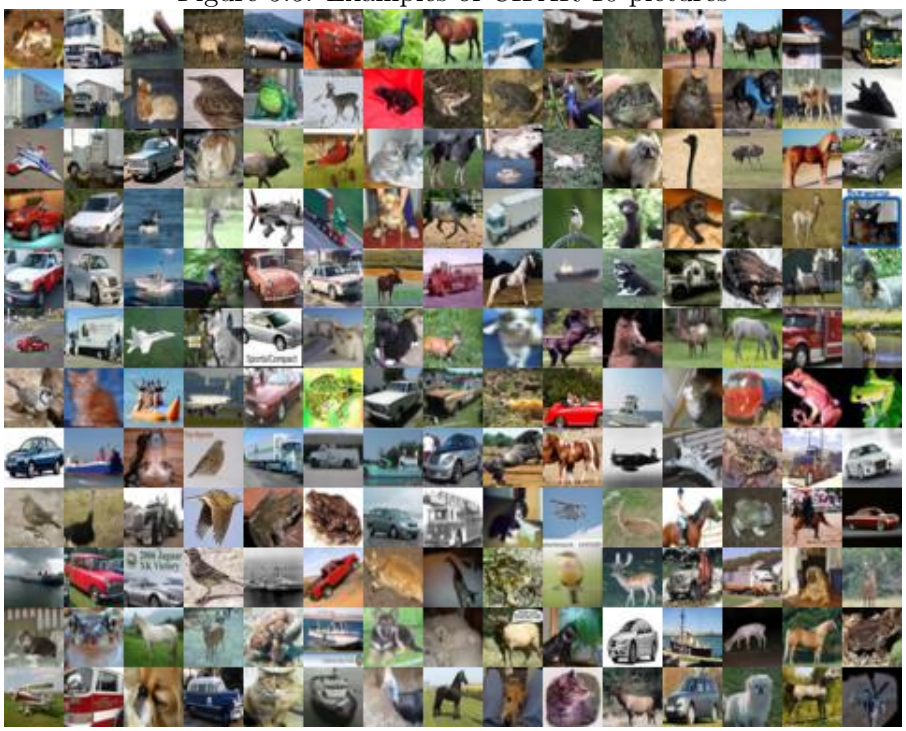
- Weight initialization: pretrained on ImageNet 2012 dataset
- Optimizer: stochastic gradient descent
- Error: softmax cross entropy with logits
- Non-linearity: ReLU

5.4. Residual network on CIFAR-10

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 of size 32×32. Each class is represented by 6,000 images. The dataset was created by Alex Krizhevsky, Vinod Nair and Geoffrey Hinton. There also exists a bigger dataset called CIFAR-100.

Figure 5.5: Examples of CIFAR-10 pictures

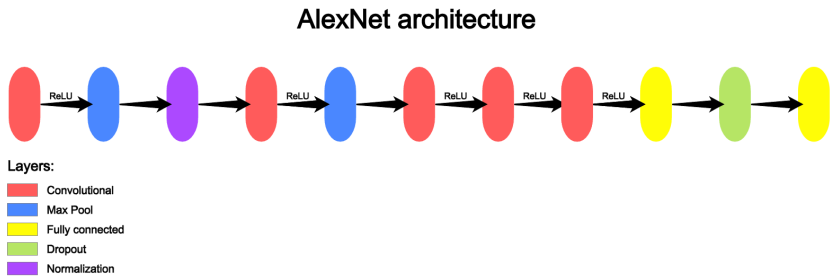


5.4.2. Network architecture

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

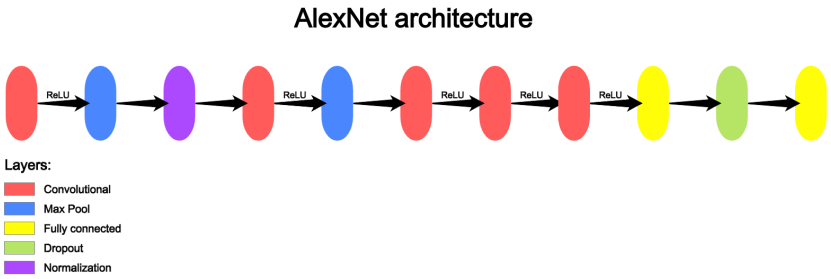
Figure 5.6: TODO schemat bloku



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
- pool

Figure 5.7: TODO schemat resneta



5.4.3. Parameters

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

Chapter 6

Implementation details

6.1. Used framework

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

All frameworks that we analyze (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 [6]. 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.

Our next tool was Jupyter notebook, which enabled us to write and share code more easily and made us more productive. We shared these notebooks and worked on them together using GitHub - a popular software development platform.

6.2. Our implementation

6.2.1. Implementation in Python

6.2.2. Implementation in C++

It is possible in Tensorflow framework to add a so-called core user-defined operation to the already 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. Source files were added to the tensorflowcoreuser_ops directory.

First part of both kernel implenetations is validation of user provided data e.g. checking whether filter's and input's depth are equal and whether the strides attributes is well-defined.

Second part is where the calculation of binarized convolution is being conducted. Its design follows the algorithms described in section **??**. 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 calculating averaged inputs accross channels and for each output's cell calculating an appropriate 'beta' parameter. Those parameters are being stored in a matrix 'K', which name was suggested in the referenced paper. Also in this case some speed-ups were implemented. In a naive implementation, calculation of matrix 'K' was doing repeated operations of calculating a sum over the same subset of input data. To avoid summing the same elements many times, a 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 shared by all available CPU cores.

Both implementations follow a typical scheme of a Tensorflow operation. They consist of two sections - in one an operation is being declared (REGISTER_OP) and in the second one the registered operation is being 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, that was compiling our source files into a shared object file, which then was 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 being created.

Chapter 7

Experiments results

7.1. Lenet on MNIST

7.1.1. Results and discussion

Results The results of experiments are in table 7.4.

Table 7.1: LeNet binarization 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 on the precision. The results are not surprising nor ground-braking. We made this experiment mostly for verifying our binarization methods.

7.2. AlexNet on Flowers

7.2.1. Binarization results and discussion

The results of experiments are in table 7.4.

Table 7.2: AlexNet on flowers-17 (50 epochs)

Accuracy	Full-precision	Binary weights	Binary weights & inputs
top-1	76%	75%	61%
top-5	98%	99%	91%

Discussion The results are promising considering the input images have high resolution. This experiment shows that fully binarized network does not lose as much precision as it was expected. Surprisngly, there is only slight difference between full-precision and binary weights network.

7.2.2. Fine-tuning results and discussion

The results of experiments are in table 7.4.

Table 7.3: Fine-tuning AlexNet on flowers-102

Full-precision	Binary weights
85%	72%

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

7.3. Residual network on CIFAR-10

7.3.1. Results and discussion

Binary Resnet XORNET style

Results The results of experiments are in table 7.4.

Table 7.4: 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 are no much difference in accuracy. That is very interesting result, because it shows that binarizing ResNet is only slightly decreasing accuracy so it is worth to use it to save memory and computation time, especially on CPU.

Binary Resnet DoReFa style

Results The results of experiments are included in table 7.5.

Table 7.5: Results of Resnet and BinResnet

blocks	layers	learning rate	epochs	ResNet	BinResnet	epochs
2	14	0.1	15	80%	78%	25
5	32	0.1	15	82%	80%	40
18	110	0.1	20	84%	80%	40
2	14	0.01	15	78%	65%	25
5	32	0.01	15	79%	67%	25
18	110	0.01	20	81%	67%	40

Discussion The network with binarization which takes average over all filters is affected more easily by changes in model parameters. For smaller learning rates it learns quite slow, achieving around 75% of accuracy compared to classical resnet in same number of epochs. It has a potencial for longer training - training it for more than 40 epochs can generate accuracy around 0.8.

Of course it is faster than XORNET implementation, but because it requires more training and gets lower accuracy, one must decide if this is good approach based on available computation machine. For testing on personal laptop, XORNET is better solution.

Binary Resnet with binary weights and activation

Results **TODO:**eksperymenty Adama

Discussion

- Implementation which is using tf.nn.conv2d 2 times in each binarization process is much slower.
- Network with binarized both weights and activation is much harder to learn. In 2-layer Lenet the results were still very stable, but adding more layers, for example 14 in 2-blocks ResNet made network able to learn only during 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 8

Conclusions

8.1. Discussion of experiments results

TODO:tu opis wspólnych konkluzji dla wszystkich eksperymentów

8.2. Future work

There are a couple of things, that could be further analyzed or added, namely:

- Right now 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 C++ are designed only for CPU. For 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 is not saving any memory (it is using as much memory as the standard convolution operation). This could be improved by adding support for one or two bit types.
- Right now it was only checked, how much the binarized network needs to be extended to achive the accuracy of its standard version. It could be also checked, 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.

Chapter 9

Team members contribution

9.1. Krystyna's contribution

- organising the team's work
- contact with project supervisor
- implementation of Resnet
- experiments on Resnet

9.2. Jakub's contribution

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

9.3. Przemysław's contribution

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

9.4. Adam's contribution

- implementation of Alexnet
- experiments on Alexnet
- code review for collegues

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] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, Yuheng Zou, *DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients*, https://arxiv.org/abs/1606.06160 (2016)

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

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