ALGORITHMS FOR SPEEDING UP CONVOLUTIONAL NEURAL NETWORKS

Doctoral Thesis

by

VADIM LEBEDEV

DOCTORAL PROGRAM IN COMPUTATIONAL AND DATA SCIENCE AND ENGINEERING

Supervisor
Professor Victor Lempitsky

Moscow

Abstract

Modern convolutional neural networks excel in many areas, but their practical application is often hindered by their high computational cost. In this work, we provide a systematic review of the literature on the topic of speeding up convolutional neural networks and present three novel methods. The first method uses a low-rank tensor decomposition of the convolutional weights to modify the neural network architecture. For the approach, we employ sparsity-inducing regularizer to prepare convolutional layers for structured pruning and develop the modified implementation of convolution that accelerates proportionally to the achieved sparsity level. The third approach performs fast fine-grained classification with the combination of a light-weight neural network with a radial basis function algorithm. The proposed methods fit into existing deep learning frameworks and allow to finetune accelerated models, leading to considerable speedups of modern convolutional architectures.

Contents

List of Abbreviations

AlexNet	a CNN architecture named after Alex Krizhevsky [Krizhevsky et al., 2012]
BLAS	Basic Linear Algebra Subprograms, a specification for low-level linear algebra libraries [Blackford et al., 2002]
CIFAR	(CIFAR10 and CIFAR100) two datasets of small images and associated classification tasks [Krizhevsky and Hinton, 2009]
CNN	Convolutional Neural Network
CP	Canonical Polyadic decomposition, a tensor decomposition
CPU	Central Processing Unit
GPU	Graphics Processing Unit, a specialized hardware for fast parallel computations
ILSVRC	ImageNet Large Scale Visual Recognition Challenge, sometimes called ImageNet, a large-scale image dataset and an associated classification task [Russakovsky et al., 2015]
MNIST	Modified National Institute of Standarts and Technology dataset, a dataset of small images and an associated classification task [LeCun et al., 1989]
NLS	Nonlinear Least Squares, an optimization method
ReLU	Rectified Linear Unit, $f(x) = \max(0, x)$, a popular activation function in neural networks
ResNet	Residual Network [He et al., 2016], a CNN architecture
SGD	Stochastic Gradient Descent, an optimization method
SqueezeNet	a compact CNN architecture [Iandola et al., 2016]
VGG	a CNN architecture named after Visual Geometry Group, University of Oxford [Simonyan and Zisserman, 2015]

Chapter 1

Introduction

1.1 Motivation

Convolutional neural networks (CNNs) are extremely powerful models which dominate modern computer vision. CNNs are used for image classification, segmentation, detection, filtering and generation tasks. Similarly, deep learning methods flourish in language processing, signal processing, and general purpose reinforcement learning.

Although the basic ideas behind CNNs date back to 1980s [Fukushima and Miyake, 1982, LeCun et al., 1989], for a long time the neural networks were not able to live up to the expectations imposed in the early days of AI research. The breakthrough for CNNs in computer vision came about only with the introduction of powerful GPUs to the learning process [Chellapilla et al., 2006, Raina et al., 2009, Krizhevsky et al., 2012].

Nowadays, GPUs are widespread in academia, and novel results are often obtained by using excessive computational power, unavailable to the authors of the previous state-of-the-art solution. New models are growing larger and slower, and this situation opens a huge gap between research and practical application. This gap manifests in several areas:

The smartphone has become a critical element of modern life, and the people are going to rely on the wearable devices even more in the future. The neural networks are among the major tools for making smartphones and wearable devices smarter; they are used for optical character recognition, face recognition, natural language processing, etc. All these problems can be solved on the server side, but the privacy concerns, time constraints or unreliable Internet connection make an offline solution more desirable. Computational capacity of a modern smartphone is remarkable compared to the previous generations of devices, but it is still no match to a GPU server, and the battery power is

also limited. Powerful CNNs would run slower on weaker hardware, and while researchers are willing to wait, end users are not so patient. Adapting CNNs to weak hardware is one of the key challenges of modern deep learning.

Autonomous driving is a rapidly developing area of research which promises to make a major impact in the near future. Autonomous driving systems often rely on multiple sensors, including radars and lidars. However, the fact that regular human equipped with a pair of eyes can drive a car means the autonomous driving system can be built with pure computer vision. The key problems before this approach are reliability and speed, as the autopilot has to react to the situation change promptly; superhuman speed is desirable. Hardware specifications may differ in this case, but conditions are not so harsh in terms of memory and electric power.

Large-scale image processing. Some applications of computer vision are characterized by the large scale of data to be processed. On example is the image retrieval, which is the problem of information retrieval with an image as a query. It can be implemented by computing descriptors of all images in the database and then comparing the descriptor of the query image with database descriptors. For the modern search engines, the database includes all the images on the Internet and thousands of queries have to be processed at the same time, which requires huge computational power. Even if a necessary number of powerful GPUs are available, faster models are still useful as a measure for the conservation of electrical energy.

In the attempt to solve these problems, a new area of research was created: acceleration and compression neural networks. Two tasks often go hand in hand and can be approached with similar methods, but in this work, we focus mainly on the acceleration problem.

The main part of this thesis describes approaches for speeding up CNNs. In the remaining introductory part, we briefly explore tasks, solved with CNNs, and list popular CNN architectures and their building blocks, used as starting points in the following chapters.

1.2 Problems and datasets

The speeding up of CNNs is relevant for all the fields of their application, including image classification, object detection, segmentation, etc. Most of the approaches described in this thesis are general, but the problem-specific approaches are also described in Section 2.8 and Chapter 5.

Image classification is considered to be the most typical task for CNNs, and most papers on the subject use classification task to demonstrate their achievements. The following datasets are often used for these demonstrations:

MNIST. The MNIST database of handwritten digits is probably the single most famous dataset in machine learning. It consists of 70000 (60000 train + 10000 test) 28×28 pixel grayscale images, each belonging to one of 10 classes. MNIST was used in the seminal papers on convolutional neural networks [LeCun et al., 1989], and still remains popular because its small size and relative simplicity allows to run experiments and achieve results quickly.

CIFAR10 and CIFAR100 are the labeled subsets [Krizhevsky and Hinton, 2009] of large unlabeled 80 Million Tiny Images Dataset [Torralba et al., 2008]. Ten classes included in CIFAR10 are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. These datasets are similar to MNIST in size (32×32 pixels), but have color, and are much more diverse. For example, an image labeled as a bird can depict a bird flying in the sky, or a close-up of an ostrich's had. Man-made objects, such as trucks and boats, can be painted in a variety of colors. This diversity makes CIFAR10 and especially CIFAR100 much more complex than MNIST.

ILSVRC2012. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [Russakovsky et al., 2015] has been running annually since 2010, although the 2012 version is the most widely used. It has become the standard benchmark for large-scale object recognition. The most challenging dataset on this list, it consists of 1.2 million images in the train set and 50000 in the validation set, of 1000 classes. The size of images varies, but they are commonly resized to 256×256 pixels. The ILSVRC dataset is a subset of ImageNet – an even bigger database with 14 million images in 21841 hierarchically organized categories), which was labeled via crowdsourcing.

Training neural networks on the datasets of this size was made possible by switching from CPU to GPU computations [Krizhevsky et al., 2012], but CNN deployment on CPU-only devices is often required in practice. Therefore, in the next section CPU and GPU performance is reported separately. While in this thesis we focus exclusively on CPU and GPU, other computational architectures can be used to speed up neural networks. These architectures include field-programmable gate array (FPGA) [Ovtcharov et al., 2015] and Application-Specific Integrated Circuits (the most notable example is Google's Tensor Processing Unit (TPU) [Jouppi et al., 2017], specifically designed for the tensorflow framework). Although these solutions provide a significant speedup in some cases, their flexibility is limited by design, an obvious downside for application in research.

ILSVRC2012 accuracy is considered to be an adequate measure of the model's performance on real-world tasks, and this dataset often becomes an end goal for general purpose CNN speed up algorithms. CIFAR and MNIST are commonly used for preliminary experiments. However, there is a drastic difference between ILSVRC2012 and these two datasets, particularly MNIST, in terms of size and complexity. This raises the question if the success in preliminary MNIST experiments is representative of performance on larger datasets. In my experience, it is not the case, as MNIST classification is too easy, and approaches that work on MNIST often cannot be scaled on larger datasets.

Other datasets which are introduced to cover this gap include Caltech-UCSD Birds dataset [Wah et al., 2011] and Stanford Cars dataset [Krause et al., 2013]. These fine-grained classification datasets contain large images, like ImageNet, but a relatively small number of classes.

1.3 CNN building blocks

In this section, we define the basic building blocks of convolutional neural networks before moving on to the description of algorithms designed to speed them up. The neural networks are organized as a stack of transformations (layers), and the key component of CNN, which gave the model its name, is the convolutional layer.

Neural networks operate on data that is organized into 2D arrays, also called maps or channels, 3D or 4D arrays. n-dimensional array is also called an nth-order tensor, although in deep learning this two terms are mixed sometimes.

The convolution in CNNs is based on the concept of linear image filtering, which was in use long before the modern era of CNNs. The linear filter takes an input 2D array (map, channel) U, applies to it a 2D filter (or kernel) and produces another 2D array V. The filtering can be defined as a convolution

$$V(x,y) = \sum_{i,j} W(x-i, y-j)U(i,j) = \sum_{i,j} W(i,j)U(x-i, y-j) \tag{1.1}$$

or as a cross-correlation

$$V(x,y) = \sum_{i,j} W(i,j)U(x+i, y+j) \tag{1.2}$$

(1.1) can be transformed into (1.2) and vice versa by flipping the filter W. In many deep learning frameworks as well as in the rest of the thesis the cross-correlation formula is used, but by tradition the corresponding layer is still called convolutional.

The limits of summation are defined by the size of filter W, which is assumed here to be $d \times d$ square. The out-of-bounds indices in U are handled by padding the input tensor, usually with zeros. This padding is usually small and does not interfere with the algorithms discussed in this thesis.

While (1.1) and (1.2) process single-channel (grayscale) images, the color images are represented as a stack of 3 channels (or maps) or a single 3D array. Likewise, the data passed between convolutional layers are also represented by 3D array, with the number of channels that is usually much larger than 3. With the introduction of a third dimension to the input U, the filter W also becomes a 3D array:

$$V(x,y) = \sum_{i=1}^{d}\sum_{j=1}^{d}\sum_{c=1}^{C} W(i,j,c)U(x+i,y+j,c) \tag{1.3}$$

Finally, application of multiple 3D filters results in multiple outputs maps, which are stacked into a single 3D output array. The filters are then organised as a single 4D array.

$$V(x,y,k) = \sum_{i=1}^{d}\sum_{j=1}^{d}\sum_{c=1}^{C} W(i,j,c,k)\,U(x+i,y+j,c) \tag{1.4}$$

This transformation of 3D arrays (or third order tensors) is called the generalized convolution. It is included in the neural networks as the convolutional layer, and the array W is treated as the parameter of the model. Alternative names for W include: convolutional weights, convolutional kernel, kernel tensor.

In convolutional layers, each neuron is connected to a small subset of neurons in a previous layer, and this subset is called the receptive field. The number of floating point operations in a convolution can be estimated as $HWCNd^2$, where H, W and C are the dimensions of input array, N is the number of filters, and d is the filter size.

Other building blocks of CNNs include:

- In **Fully-connected layer**, each neuron is connected to all neurons of the previous layer. This layer takes an input vector x with C elements, multiplies it with the weights matrix $W \in \mathbb{R}^{C\times N}$ producing an output vector y with N elements, in CN floating point operations. If the input is not a vector, it is reshaped and treated as a vector.

- **Nonlinearity**. Linear layers, such as the convolutional and the fully-connected layer, are interleaved with nonlinearities. The most popular nonlinearity is the rectified linear unit (ReLU) function $f(x) = \max(0, x)$. This computationally cheap operation is applied element-wise, so its cost is negligible compared to other components of CNNs. Since nonlinearity is almost always present after the fully-connected or convolutional layer, it is often omitted in the architecture description.

- **Pooling**. Pooling layer subsamples inputs with maximum, average, or another kind of aggregation method. Downsampling image s times reduces the number of operations in subsequent layers s^2 times, making the proper positioning of pooling layers one of the critical decisions for building fast CNNs. As an alternative to separate pooling layer, downsampling can be performed in the convolutional layer with stride bigger than one.

- **Batch normalization** [Ioffe and Szegedy, 2015]. BatchNorm layer normalizes inputs to zero mean and unit standard deviation. An introduction of batch normalization can drastically speed up training convergence and improve the final result. Normalization is as ubiquitous as nonlinearity in modern architectures, so it also can be omitted in the architecture schemes.

Convolutional layers usually have the largest operation count and consume the most of memory and time in CNNs, as demonstrated in Figure 1.2. Thus, the convolutions are in the main focus of this work.

Several specific cases and modifications of general convolution, used to save build faster models are shown in Figure 1.1 and described in detail below.

- 1×1 **convolution**. Convolution complexity decreases for smaller spatial sizes of the kernel, and smallest possible size is 1. In this case convolution (1.1) reduces to a linear combination of input channels:

$$V(x, y, k) = \sum_{c=1}^{C} W(c, k)\, U(x, y, c) \tag{1.5}$$

  On top of small operation count, this operation can be efficiently implemented by matrix multiplication.

- **Group convolution.** Another way to reduce the cost of convolution is to cut some of the connections between input and output channels. The idea is implemented by dividing input and output channels into several groups G_k and cutting all the

connections between different groups:

$$V(x, y, k) = \sum_{i=1}^{d} \sum_{j=1}^{d} \sum_{c \in G_k} W(i, j, c, k)\, U(x + i, y + j, c) \tag{1.6}$$

Group convolution was originally proposed by [Krizhevsky et al., 2012] as a way to construct a model what can be parallelized between two GPUs, and later this concept was revived as a building block of fast CNNs.

- **Depthwise convolution.** If the number of groups is the same as the number of input and output channels, the convolution is called a *depthwise convolution*. In this case, every channel is filtered independently by a single filter:

$$V(x, y, k) = \sum_{i=1}^{d} \sum_{j=1}^{d} W(i, j, k)\, U(x + i, y + j, k) \tag{1.7}$$

The number of channels in the output array is fixed in this case to the number of channels in the input array. Depthwise convolution requires C times less floating point operations compared to a regular convolution of the same size, but the actual timings depend on the efficiency of implementation.

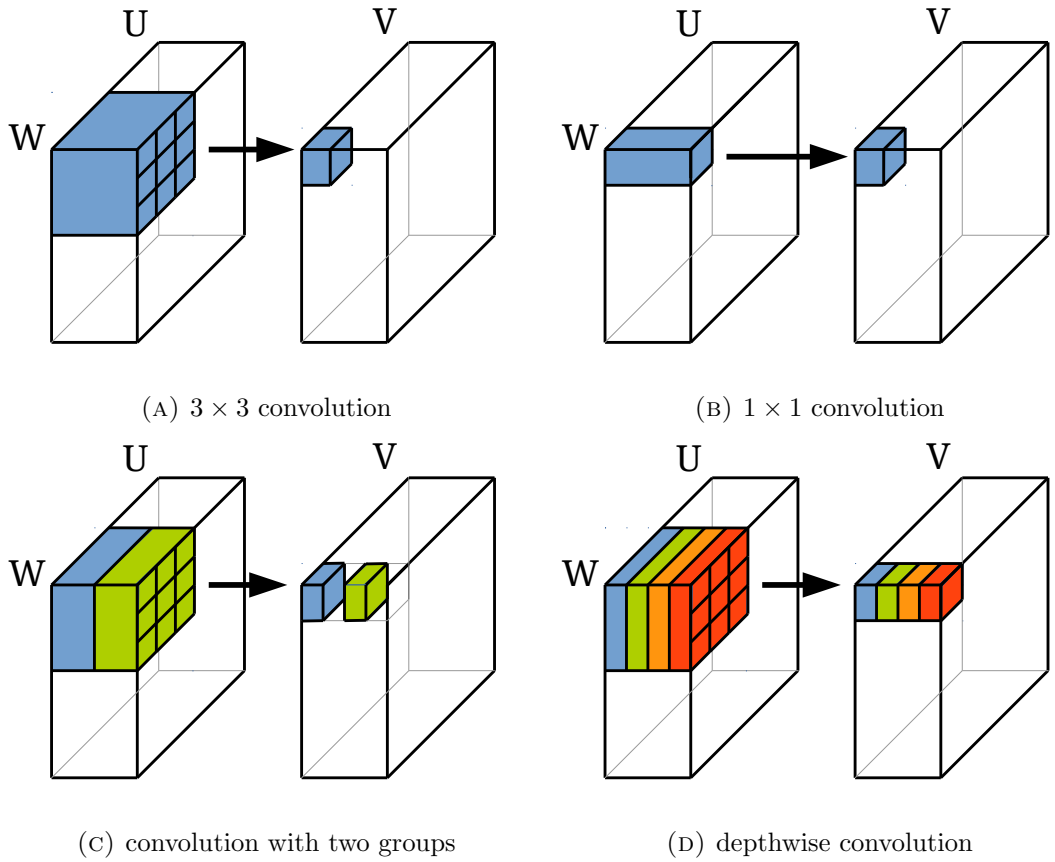
In a practical setting, then the available CNN does not fit the speed constraints, there are several options to be exhausted:

1. Use more powerful hardware.
2. Pick more efficient implementation of convolution operation.
3. Tune the model or use fast approximations.

Switching to the new hardware (which includes a transition from CPU to GPU computation) is the simplest option, as in modern deep learning frameworks it does not require any programming. The increase of processing speed and memory size in modern GPUs is one of the main factors pushing the capabilities of neural networks.

The details of implementation are also hidden in modern frameworks. Although questions of hardware and implementation are mostly outside of the scope of this work, some the aspects of implementation may influence the design of approximate speed-up methods.

The naive implementation of convolution operation (1.4) has multiple nested loops, in general case six of them. According to Chellapilla et al. [2006], small kernel sizes make



(A) 3×3 convolution (B) 1×1 convolution

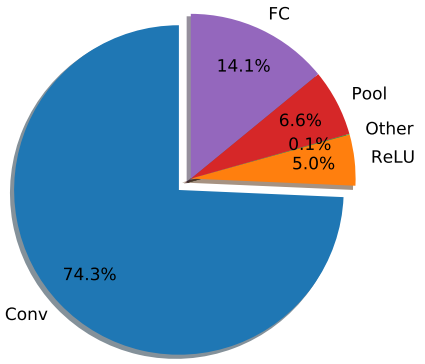
(C) convolution with two groups (D) depthwise convolution

FIGURE 1.1: Variants of convolution used in modern CNN designs. (A) Standard convolution with 3×3 filters. (B) 1×1 convolution, which rearranges input maps and does not capture relations of neighboring pixels. (C) Both input and output maps are divided into two groups (indicated by blue and green), with no connections between them. (D) In depthwise convolution, all maps are processed independently.

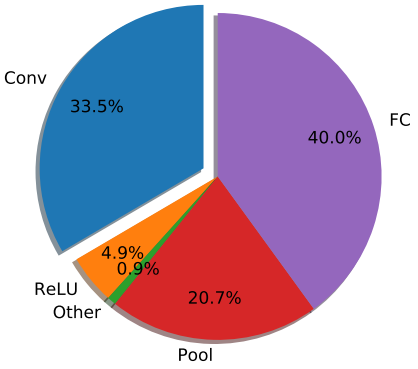
the inner loops very inefficient as they frequently incur JMP instruction. Additionally, forward and backpropagation steps require both row-wise and column-wise access to the input and kernel, a feature that cannot be implemented efficiently with common data representations.

The issues of naive implementation are addressed by Chellapilla et al. [2006] with an approach called unrolled convolution. The central idea is to reduce convolution to the multiplication of two matrices by duplication of input data. The reduction allows using highly optimized implementations of matrix multiplications (variants of BLAS [Blackford et al., 2002] libraries) that have been developed over many years for different computing architectures, including CPU and GPU. The reduction is demonstrated in Figure 1.3.

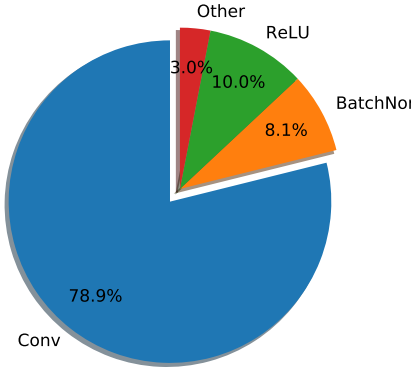
The construction discussed above has proven to be highly successful and is used in the majority of modern CNN frameworks, e.g. [Chellapilla et al., 2006, Donahue et al., 2014, Jia et al., 2014a, Chetlur et al., 2014, Vedaldi and Lenc, 2014, NervanaSystems, 2015].



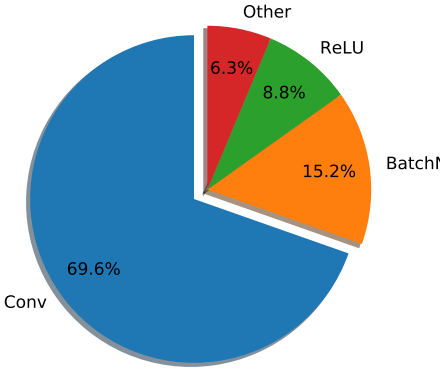
(A) AlexNet CPU timings



(B) AlexNet GPU timings



(C) ResNet-50 CPU timings



(D) ResNet-50 GPU timings

FIGURE 1.2: Timings of different layers for AlexNet and ResNet-50 architectures on CPU (Intel Core i7-6800K) and GPU (GeForce GTX 1080), measured in Pytorch framework by a built-in profiler. Surprisingly, the modern implementation of convolution on GPU is so efficient that for relatively shallow architecture such as AlexNet, the largest part of running time is spent in fully-connected layers. In case of CPU as well as for deeper architectures, bulk of the time is consumed by convolutional layers. In the ResNet-50 example, the single fully-connected layer of this architecture takes less than 0.5% of the total running time both on CPU and GPU.

Another way to build a faster implementation for convolution, explored by Mathieu et al. [2014], is based on Convolutional Theorem which states that circular convolutions in the spatial domain are equivalent to pointwise products in the Fourier domain. Denoting Fourier transform as $\mathcal{F}$ and inverse transform as $\mathcal{F}^{-1}$, we can express convolution of two 2D maps f and g the following way:

$$f * g = \mathcal{F}^{-1}\left(\mathcal{F}(f)\mathcal{F}(g)\right) \tag{1.8}$$

The main benefit of this approach is that its computational complexity does not depend

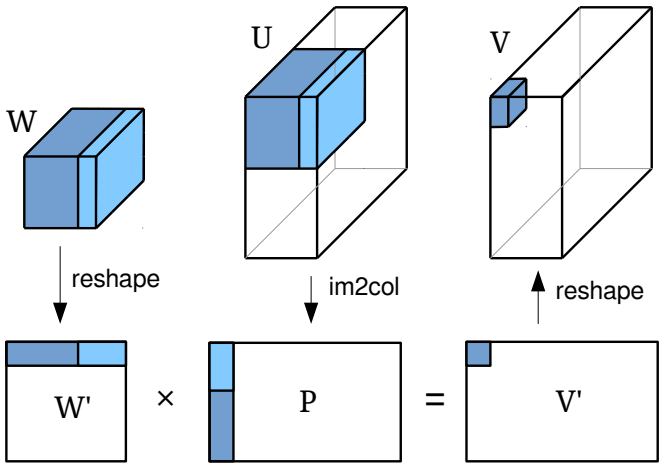


FIGURE 1.3: Reducing convolution to matrix multiplication through unrolling. Input array U is transformed to patch matrix P by the im2col operation. Columns of P are built from unraveled patches from U, with patch size defined by the size of the filters in weight array W. The patch matrix P is then multiplied by the weight matrix W' (obtained from W by reshaping), resulting in output matrix V'. The final output array V is obtained from V' by another reshape. Highlighted in blue are one patch in U with the corresponding column of P, filter in W with the corresponding row of W' and single output pixel produced by this filter in this patch. Not that only of the filters of W is drawn, as it is hard to visualize four-dimensional array.

on the filter size, which is beneficial for larger filters. The disadvantages are larger memory requirements for storage of feature maps and filters in the Fourier domain, and possible slow down in case of small filters. Since filters used in modern architectures are mostly small, this method is not very common.

Other efficient approaches for implementation of convolutions with small kernels include usage of Strassen fast matrix multiplication algorithm [Cong and Xiao, 2014] and Winograd minimal filtering algorithm [Lavin, 2016].

1.4 CNN architectures

In this section, we list several popular architectures that are used further in the text.

LeNet [Lecun et al., 1998] is a simple CNN architecture initially proposed and still often used for the MNIST dataset. It consists of two convolutional, two pooling and two fully-connected layers. Original architecture included tanh nonlinearities and RBF units which are usually replaced by ReLU and regular fully-connected layer in the modern implementations of this architecture.

AlexNet [Krizhevsky et al., 2012] was first CNN successfully trained on a large-scale dataset, a breakthrough which led to the victory at ILSVRC2012 competition. Being the first, AlexNet had some peculiar features: filters of varying sizes, including large filters on the first layers, and relatively low depth. Although modern CNNs exceed AlexNet in every aspect, it is still often used as a common baseline. AlexNet is fast compared to the deepest and most accurate of advanced CNNs, but not compared to the fastest architectures on the same accuracy level.

Early CNN architectures used large convolutional filters, such as 5×5 filters in AlexNet and LeNet, and AlexNet even had 11×11 filters on the first layer. This large size allowed Krizhevsky et al. [2012] to observe smoothness of trained filters which indicates that much less is required to define the filter with the help of interpolation or some other procedure. The similarity of several filters to vertical or horizontal edge detection filters points to the specific method: separable filters. The works on separability and extension to tensor decompositions are reviewed in Section 2.1.

VGG is a family of CNNs defined by two key features: considerable depth and exclusive use of 3×3 filters, which is the smallest size to capture the notion of left/right, up/down, center [Simonyan and Zisserman, 2015]. The success of VGG architectures launched the increasing depth of modern CNN architectures: the more layers you can stack, the better. VGG architectures are very slow and even heavier in number of parameters, mostly due to massive fully-connected layers. VGGNets are still popular among researchers because of their simple structure, and even dominate some applications, such as fine-grained classification and image stylization.

ResNet. As noted by He et al. [2016], some limit of CNN's depth still exists: when the network becomes deeper, its accuracy saturates and after reaching some limit starts rapidly degrading. This problem is not caused by overfitting, but by the failure of the training process. A novel design approach was proposed to facilitate easier gradient propagation through the network and therefore help the training process. The main idea is to organize CNN's blocks as a residual function

$$f(x) = h(x) + x \tag{1.9}$$

where x is the input and $h(x)$ is a block of convolutional layers. Residual training overcomes accuracy degradation problem and allows to efficiently train CNNs up to thousand layers deep, although these extreme sizes are only possible on the datasets with small input sizes, like CIFAR. Members of the ResNet model family are designated by the number of layers in the network, with medium-sized ResNet-50 being the most widely used model. Attempts to improve original ResNet architecture include ResNeXt [Xie

et al., 2017] which introduces group convolutions to residual block, and DenseNet [Huang et al., 2017], which creates additional connections between residual blocks.

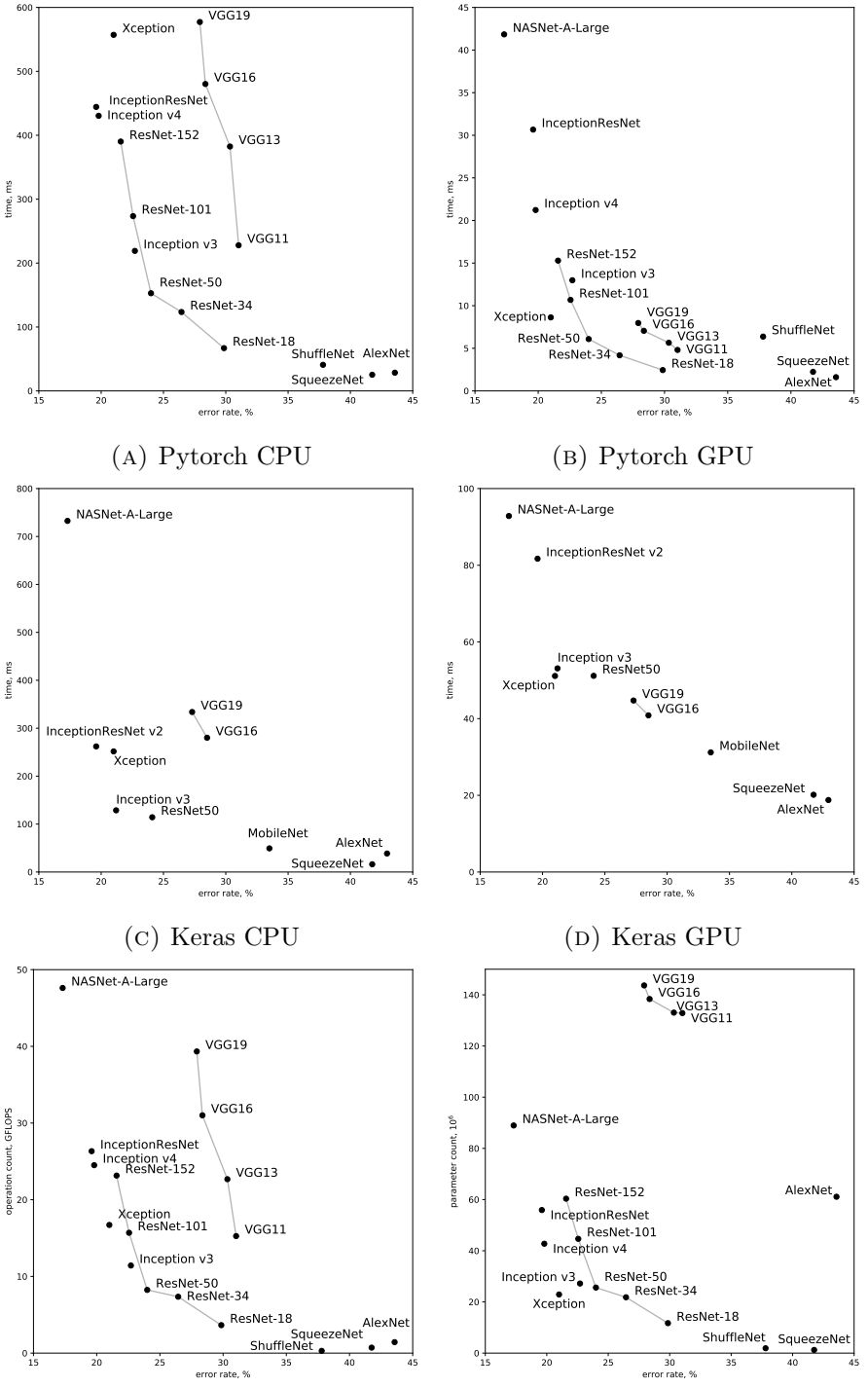
Inception family also known as GoogLeNet [Szegedy et al., 2015]. The main idea of the Inception architecture is based on finding out how an optimal local sparse structure in a convolutional vision network can be approximated and covered by readily available dense components. This principle led to the construction of Inception block, which consists of several branches, each with filters of a specific size. This block is repeated several times, resulting in a very deep neural network. Nowadays, the inception family includes four versions of Inception model [Szegedy et al., 2015, 2016, 2017], the Xception [Chollet, 2017] model which uses depthwise separable convolutions and hybrid model called Inception-ResNet [Szegedy et al., 2017].

Accuracy is prioritized over speed and compactness in the mainstream CNN research, which has been moving in the direction of increasing CNN depth for some time. Nevertheless, training of the deepest of modern CNNs is not possible without paying attention to the efficiency of designed architecture. The principles of efficient architecture design and CNNs designed for maximal speed are covered in detail in Section 2.2.

The correspondence between inference time and accuracy of described CNN architectures is shown in Figure 1.4. Four charts compare CNN performance on CPU and GPU on two frameworks, Pytorch and Keras with Tensorflow backend, and the operation counts are shown on the fifth chart. This comparison reveals the influence of hardware and software details on the relative performance of different architectures, especially if the network uses non-standard convolution type, such as group convolution or depthwise convolution. This phenomenon can be observed with the Xception architecture, which drastically changes its position relative to the neighboring models, e.g. ResNet-50. These changes occur not only with the framework and CPU/GPU switch but also between the different versions of the same framework and different GPU models. Although these factors are extremely important in practice, we leave most of the details of hardware and software implementation outside of the scope of this work, and focus on the algorithms and approximation ideas.

Chronologically first model, AlexNet, lies in the bottom right corner in all versions of the chart, meaning it is among the fastest and least accurate models. The next generation models, such as VGG, ResNets and various models from Inception family, occupy the center and left parts of the diagram. The optimal models lie on the lower convex envelope of this diagram. The general target of speeding up neural networks is to push this envelope down and to the left. In my experiments with the Pytorch framework, the central part of this envelope includes ResNet models and Inception family models. VGG models in all cases with the exception of GPU computation with Keras, are situated

higher, which means they are not optimal. The top left part of the envelope corresponds to the most accurate models which trade a lot of speed for accuracy. The slowest and most accurate among models shown on this chart is NASNet, which is created with automatic architecture search, an approach described in Section 2.3.



(A) Pytorch CPU (B) Pytorch GPU

(C) Keras CPU (D) Keras GPU

(E) operation count (F) parameter count

FIGURE 1.4: The trade-off between the inference time and the ILSVRC Top-1 classification error for some of CNN architectures. The timings are measured for both CPU (Intel Core i7-6800K) and GPU (GeForce GTX 1080) with two frameworks: Pytorch 0.3 and Keras with Tensorflow backend. Additionally, the operation and parameter counts for Pytorch models are presented on the lowest chart. Lines connect groups of similar architectures. NASNet-A-Large architecture is not shown on the Pytorch CPU chart as its inference time in this setting was measured at 2.2 seconds, which puts it too far away from the rest of the points.

1.5 Contribution

The thesis has the following contributions:

- We describe a novel CNN speedup algorithm based on low-rank CP-decomposition of convolutional weights. We show what CP-decomposition can be used to replace one convolutional layer with four smaller layers, which produce approximately the same output significantly faster. We implement this method with existing CNN building blocks so that it can be efficiently incorporated into existing deep learning frameworks, and, most importantly, the decomposed version of the network can be finetuned to regain accuracy drop inflicted by approximation.

  We evaluate the idea on small optimal optical character recognition task and ILSVRC dataset and obtain competitive results. These findings are presented in Chapter 3 and published as [Lebedev et al., 2015].

- We analyze the implementation of a convolutional layer and discover an opportunity to perform sparse convolution without overhead costs usually associated with sparse operations. My method uses structured sparsity, essentially changing filter shapes with special constraints. Then, we impose structured sparsity on the neural network by training with sparsity-inducing regularizer and pruning. The method is evaluated and carefully compared with baselines on MNIST and ILSVRC datasets, and state-of-the-art results are obtained. Additionally, we demonstrate trained sparsity patterns and show what the training process prefers circular filters in the wide range of training conditions. This contribution is the subject of Chapter 4 and is published as [Lebedev and Lempitsky, 2016].

- We introduce impostor networks, an architecture that allows performing fine-grained recognition with high accuracy by combining a light-weight CNN with radial basis function (RBF) classifier. We develop three methods for joint training of two parts of the model, and carefully compare them on a variety of fine-grained classification datasets. Impostor networks are suitable for resource-constrained platforms, but it is not their only advantage. Particularly, we demonstrate the reliability of impostor nets in open set scenario, i.e. the situation when the model is presented with a sample from the class not included in the training set. This contribution is the subject of Chapter 5 and is published as [Lebedev et al., 2018].

Full list of publications:

1. Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. International Conference on Learning Representations (ICLR), full conference paper, 2015.
2. Vadim Lebedev and Victor Lempitsky. Fast ConvNets using group-wise brain damage. Computer Vision and Pattern Recognition (CVPR), 2016
3. Vadim Lebedev and Victor Lempitsky. Speeding-up Convolutional Neural Networks: A Survey. Bulletin of the Polish Academy of Sciences: Technical Sciences, 2018.
4. Vadim Lebedev, Artem Babenko and Victor Lempitsky. Impostor Networks for Fast Fine-Grained Recognition. Arxiv preprint, 2018.

The following works describe related material that has not been included in the thesis.

1. Dmitry Ulyanov, Vadim Lebedev, Andrea Vedaldi, Victor S Lempitsky. Texture Networks: Feed-forward Synthesis of Textures and Stylized Images. International Conference on Machine Learning (ICML), 2016.
2. Oleg Grinchuk, Vadim Lebedev, Victor Lempitsky. Learnable visual markers. Neural Information Processing Systems (NIPS), 2016.

Chapter 2

Related work

This chapter provides a literature review of algorithms for speeding up neural networks, focusing on approaches that aim at building CNNs that are fast at inference time. In general, these approaches can be divided into six groups:

- The approaches using tensor decompositions of the weights or activations.
- Methods relying on low-precision arithmetic and other quantization techniques. This group includes methods that aim to build fully binarized neural networks.
- Approaches that prune the weights of larger networks to build their smaller and faster equivalents.
- Teacher-student approaches, which train small networks with the aid of bigger ones.
- Efficient architectures design: methods that pursue a heuristic-driven search for the smallest possible network that can be trained from scratch to an appropriate level of accuracy.
- Methods for automatic architecture search that aim to replace human-suggested heuristics with an algorithm that designs neural networks automatically.

Below, each group is presented within a separate section. Factorization, quantization, and pruning approaches can all be grouped into a "gradual" speed-up super-group. Such methods start with a pretrained CNN and then interleave the transformation of convolutional layers with fine-tuning operations. Each transformation step leads to a speed-up as well as to the drop in accuracy. The subsequent fine-tuning operation recovers part of the accuracy drop. On the other hand, methods aimed at fast architecture design (including those that do this in an automated way) aim to design an architecture that

can be trained from scratch. Finally, the teacher-student methods take the middle path, as they usually create the final architecture in a two-step process, which first trains a slow teacher network and then trains a fast student network using the guidance from the teacher.

2.1 Tensor decompositions

The convolution layers, which correspond to the bulk of the inference time in modern CNNs, are based on the generalized convolution operation:

$$V(x, y, k) = \sum_{i=1}^{d} \sum_{j=1}^{d} \sum_{c=1}^{C} W(i, j, c, k)\, U(x + i, y + j, c)\,, \tag{2.1}$$

where U denotes the input 3D array containing C 2D image maps, V denotes the output 3D array containing N 2D image maps, and W is a four-dimensional weight array.

The convolutional layer is thus defined by its four-dimensional weight arrat W. The idea behind tensor decomposition methods is to decompose this array into a product of low-dimensional tensors, resulting in several fast convolutions with fewer operations. Filter decomposition for speeding-up convolutions was initially developed not in the context of CNNs. [Rigamonti et al., 2013] considered denoising tasks and approximated convolutional filters by a shared set of separable filters, leading to substantial speedups with minimal loss in denoising accuracy.

The approach based on separable filters was then extended to convolutional layers of CNNs in [Jaderberg et al., 2014b]. The convolutional weights with square $d \times d$ filters are approximated and replaced by a product of two tensors with $1 \times d$ and $d \times 1$ filters and K feature maps between them. The parameter K (the decomposition rank) regulates the speed-accuracy trade-off: small K leads to fast but inaccurate models, while large K allows to reproduce original convolution closely but with high computation time. Carefully tuning K for every approximated layer is a crucial part of speeding up the neural network within this algorithm. Jaderberg et al. [2014b] describes two optimization approaches, depending on whether the objective minimized by the decomposition procedure measures the error of the filter reconstruction or the error of the unit activation reconstruction. Of the two approaches, the latter is more practical as it can be made a part of the CNN fine-tuning process, which optimizes the training loss used to train CNN over all parameters of the network (although end-to-end fine-tuning was not pursued).

method	operations
full convolution	CNd^2
two-component[Jaderberg et al., 2014b]	$Kd(C+N)$
monochromatic approximation[Denton et al., 2014]	$CC_1 + Nd^2$
biclustering+svd[Denton et al., 2014]	$CC_1K_1 + C_1N_1K_1K_2d^2 + NN_1K_2$
biclustering+outer product[Denton et al., 2014]	$K(CC_1 + C_1N_1d^2 + N_1N)$
cp-decomposition[Lebedev et al., 2015]	$K(C + 2d + N)$
cp-decomposition[Astrid and Lee, 2017]	$K(C + d^2 + N)$

TABLE 2.1: Per-pixel operation counts for different approximations of convolutional layers, with the focus on CP-decomposition approaches. The number of operations depends on the size d of the square filters, the number of input channels C and the output channels N, as well as the hyper-parameters specific to approximation methods: the decomposition rank K, and the number of clusters C_1 and N_1 for input and output channels for the clustering methods from [Denton et al., 2014]

Originally, the separability was enforced on the pretrained network only, but [Jaderberg et al., 2014b] also note that low-rank filters can be learned in a discriminative manner, i.e. from scratch and at the same time with the rest of the network. This idea was later incorporated into Inception architectures, starting from the second version [Szegedy et al., 2016].

Several weight tensor compression methods based on clustering were proposed by [Denton et al., 2014]. The starting idea is to cluster the tensor slices along one or two of the four dimensions (referred to as biclustering), to split up the tensor according to cluster boundaries and then to approximate resulting slices by centroid values ("monochromatic approximation"), via the SVD decomposition, or via the canonical polyadic (CP) decomposition obtained by greedy approach (outer product decomposition). The CP decomposition is one of the generalizations of the SVD decomposition to higher-order tensors (a review [Kolda and Bader, 2009] on such decompositions is highly recommended). The splitting reduces decomposition ranks required for good approximation, but this comes at the cost of increasing the number of tensors that need to be approximated. Another shortcoming of this approach is in a rather large number of hyper-parameters, which makes it increasingly hard to tune for optimal performance.

Another algorithm that applies the CP-decomposition to the convolutional weights is described in [Lebedev et al., 2015] and Chapter 3. Instead of clustering, the focus of this work is on tuning the performance of CP-decomposition, which is done in two phases. The first phase starts with better initial approximation obtained by the non-linear least squares algorithm (instead of the greedy one used in [Denton et al., 2014]). The second stage finetunes the whole network after the decomposition is applied.

Finetuning the model after decomposition is non-trivial, especially if the decomposition is applied to several layers. Even if the single layer is decomposed, numerical instabilities

inside the four convolutional layers introduced by CP-decomposition often lead to the explosion of gradients during fine-tuning. The key problem is that CNN block designed to follow the structure of tensor decomposition does not have non-linearities between the convolutions. When multiple layers are decomposed, the fine-tuning can be done either once after all the decompositions are applied, or iteratively, after every single decomposition. The iterative approach was explored by by [Astrid and Lee, 2017], who also argues for Tensor Power Method [Allen, 2012] to be the most appropriate for obtaining initial decomposition. Training from scratch with the architecture closely related to one obtained with CP-decomposition was implemented in [Jin et al., 2014].

The work [Zhang et al., 2016] suggests an alternative way to speed up neural networks by applying the low-rank assumption to activations rather than to the weight tensor. This assumption splits one convolutional layer in two, while the weights of each of the two new layers are obtained by solving optimization problems. The focus on activations instead of the weights has two advantages: first, the nonlinearities can be taken into account in the optimization problem formulation, and secondly, in case of multiple layers approximations, the activations of the original network can be used as a target. This idea, called the *asymmetric reconstruction* limits the accumulation of error from layer to layer. Finally, as in the previous approaches, the whole network can be fine-tuned.

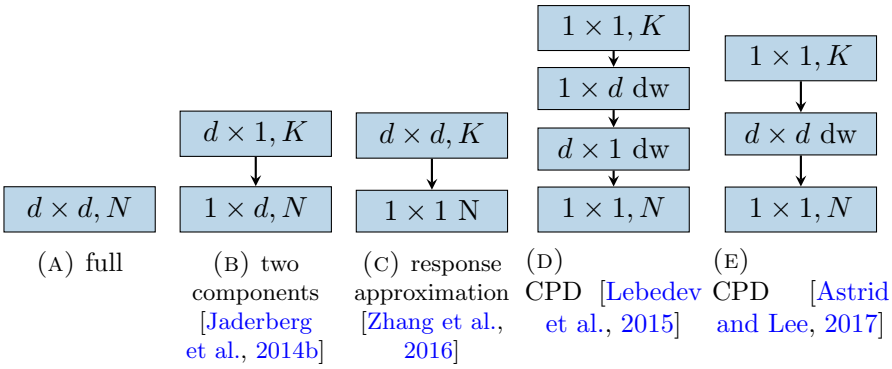
All the methods listed above replace single convolutional layer by a block of smaller convolutions. The comparison of these blocks is presented in Figure 2.1.

Other higher-order tensor decompositions have been used for CNN speed-up. Tucker decomposition was applied for speeding up and compression of CNNs in [Kim et al., 2015] and [Wang and Cheng, 2016]. Tensor Train (TT) decomposition was applied to fully connected layers of convolutional neural networks by [Novikov et al., 2015]. The main focus of that work is compression, not the speed-up, but the achieved compression rates of up to 200000 times are extremely impressive.

2.2 Fast Architecture Design

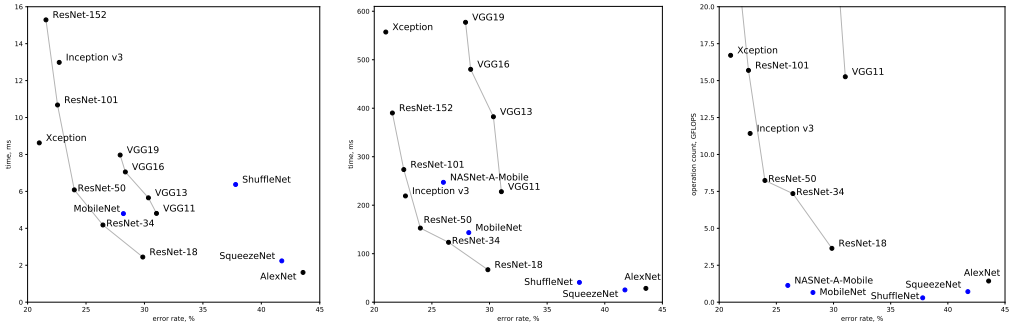
The research in CNNs has led to the emergence of several popular families of the architectures. Historically, the search for architectures was driven by the desire to push the classification accuracy (most importantly in the annual ILSVRC [Russakovsky et al., 2015] challenge), while the inference speed was of a secondary concern.

The tensor decomposition approaches are closely related to the task of designing optimal architectures, which is the topic of this section. The methods, described in this section train the designed architectures from scratch, and the design choices are often directly



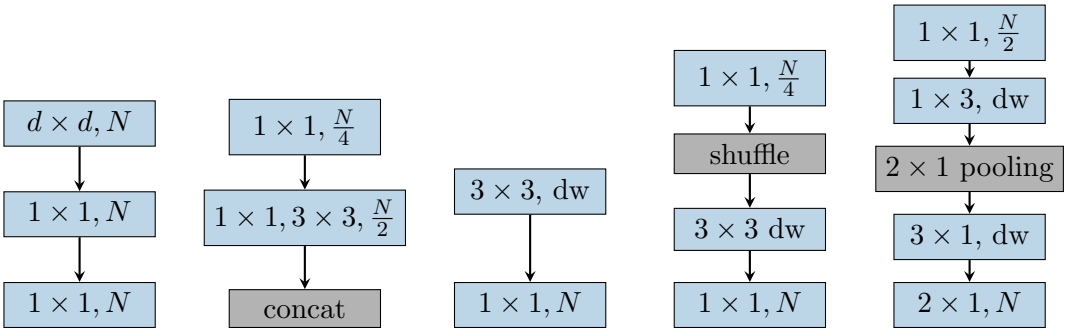
(A) full (B) two components [Jaderberg et al., 2014b] (C) response approximation [Zhang et al., 2016] (D) CPD [Lebedev et al., 2015] (E) CPD [Astrid and Lee, 2017]

FIGURE 2.1: CNN blocks used by tensor decomposition methods to replicate a single convolutional layer. Each layer here is labeled with its kernel shape and the number of filters. 'dw' stands for depthwise convolution, in which case the number of the channels in the input is the same as on the output.



(A) GPU (B) CPU (C) operation count

FIGURE 2.2: Diagram of ILSVRC accuracy and inference time for fastest of available CNN architectures. The timings are measured both on CPU (Intel Core i7-6800K) and GeForce GTX 1080 GPU with Pytorch 0.3. Operation count, the hardware-independent measure of model complexity is also shown on the right. Efficient CNNs described in this sections are marked with blue points. NASNet-A-Mobile is excluded from the left chart because it's inference time on GPU is too high compared to other models. According to the claims of the authors, efficient models such as MobileNet and ShuffleNet are supposed to be faster compared to regular models such as AlexNet. In this experiment, fast models appear to be fast only in terms of operation count, but not in the actual execution time. This result is probably caused by a relatively inefficient implementation of depthwise separable convolution in Pytorch.



(A) NiN (B) SqueezeNet (C) MobileNet (D) ShuffleNet (E) EffNet

FIGURE 2.3: Sequences of convolutional layers used for fast and compact architectures described in Section 2.2. The notation follows Figure 2.1. In case of ShuffleNet, 1×1 convolutions are also group convolutions.

influenced by preceding works on tensor decomposition, such as in the case of [Jin et al., 2014].

One of the first prominent attempts at building an architecture, which emphasizes efficiency is the Network-In-Network (NIN) architecture proposed in [Lin et al., 2014]. The basic idea behind NIN is to replace non-linearities within the convolutional network with a more complex function. Multilayer (two-layer) perceptron, which is known to be a universal approximator, is chosen as a replacement. By sharing the weights of the perceptrons across spatial dimensions one ends up with two 1×1 convolutional layers interleaved with standard non-linearities.

The SqueezeNet [Iandola et al., 2016] is a compact architecture that achieves AlexNet-level performance with $50\times$ fewer parameters. The ideology behind SqueezeNet is based on three principles:

1. Utilize 1×1 filters instead of 3×3 filters whenever possible. Smaller filters have fewer parameters and need fewer operations.
2. When 3×3 filtering has to be applied, minimize the number of input channels.
3. Downsample late in the network to give most layers chance to work with large high-resolution maps.

The first and the second principles ensure that the model is both small and fast, while the third design principle boosts the accuracy. All three principles combined yield a very small model with a slightly smaller inference time than AlexNet (which serves as the base model).

The MobileNet architecture [Howard et al., 2017] is prominent among CNN architectures designed for optimal size and inference time. The main idea is to separate filtering and feature construction functions of a convolutional layer into two layers: the depthwise convolution and 1×1 convolution (i.e. a pointwise convolution). This combination is called a depthwise separable convolution and was first popularized in [Xie et al., 2017].

The MobileNet also employs simple but effective tricks to control architecture performance: the network width (the number of channels) is controlled by the width multiplier α, and the input resolution is controlled by the resolution multiplier ρ. In general, changing the network width and the input resolution is a simple way to control trade-offs between the speed and the number of parameters on the one hand and the accuracy on the other. Such knob, however, had not been thoroughly investigated in research papers prior to [Howard et al., 2017].

In comparison with SqueezeNet, MobileNet is capable of achieving higher accuracy with approximately same model size and one-tenth of mult-add operations. This advantage in operation number, however, is hard to translate into actual inference speed-up on GPU because depth-wise convolution is not as efficiently implemented on GPU as a regular convolution.

Still, separating the depth-wise and intra-channel convolutions has become a popular idea. In modern architectures, this idea leads to 1×1 convolution being dominant in the total computation cost of the model. The only way to squeeze 1×1 convolutions even further is to turn them into *group convolutions* [Krizhevsky et al., 2012], which only mix channels within certain groups of channels. Such grouping, however, would mean that the whole network is divided into thin columns with no connection between them. ShuffleNets [Zhang et al., 2017] address this problem by using the channel shuffle operation. Shuffling of channels between the convolutions allows enjoying the low costs of group convolutions without splitting the network into disjoint parts. The ShuffleNet architecture uses an even smaller number of operations for the same accuracy level compared to MobileNet. Whether this advantage translates to actual timings depends on the efficiency of the available implementations of depthwise and group convolutions.

Another variation of MobileNet building block is the EffNet architecture [Freeman et al., 2018], motivated by a careful study. EffNet utilizes depthwise separable convolutions and pushes it even further by splitting 3×3 convolution into pair of convolutions with 1×3 and 3×1 kernels. The downsampling along one dimension is performed with strided convolution, and along other - via 2×1 pooling.

The final logical step in the movement towards smaller filters in CNNs would be a complete rejection of convolutions with filers larger than 1×1. The problem with this kind of CNN is that adjacent pixels would not be connected, and the receptive fields will always be 1×1. ShiftNets [Wu et al., 2017] solve this problem using channel shifts, which allows adjacent pixels in different channels to connect through 1×1 convolutions. Channel shift is a cheap operation, but ShiftNet often requires to increase the number of channels in the network to achieve the same level of performance. The proposed building blocks of ShiftNet and other architectures listed above are shown in Figure 2.3. Figure 2.2 shows runtimes, operation counts, and the ILSVRC accuracies of the described architectures.

The principles of lightweight architecture design are also beneficial for the design of large "heavy-weight" architectures. In a practical setting, the memory on the GPU and the time for the experiments are always limited, so the depth of the CNN architecture is limited as well. Since the CNN performance usually increases with depth, it is desirable to stretch this limit by designing architectures efficiently. In this arena, the Inception

architecture [Szegedy et al., 2015] is built on the premise of approximating sparsity with existing building blocks. It was gradually refined [Szegedy et al., 2016] to minimize the computational cost and to maximize the performance. Towards this end, convolution decomposition was introduced in the fourth version [Szegedy et al., 2017]. Finally, depth-wise convolutions were introduced to the Inception architecture in [Chollet, 2017]. The ResNet architecture [He et al., 2016] (as well as its further development ResNeXt [Xie et al., 2017]) also makes use of similar efficient architecture principles. The training of extremely deep CNNs (150 layers for ImageNet-sized inputs) simply cannot be done without careful management of computation resources.

Finally, we note once again, that efficient architecture design is not limited to classification tasks, as segmentation and detection require specialized architectures that nevertheless may share the same principles. This subject is beyond of the scope of this review, but good benchmarks that track both speed and accuracy are provided for the Cityscapes dataset [Cordts et al., 2016] in the case of segmentation and the KITTI benchmark [Geiger et al., 2012] in the case of 2D and 3D detection.

2.3 Automatic architecture search

Usually, the neural network architectures are hand-constructed by human experts, who are guided by general principles of efficient architecture design. A lot of choices in the architecture construction are left to intuition and guessing. The situation asks for automatic algorithms for architecture search that are reviewed in this section, naturally expanding Section 2.2. As automated architecture search is a rapidly developing field at the spearhead of modern deep learning research, this section only covers the most influential works in this sub-field.

Automatic architecture search is essentially a hyperparameter optimization, which is a general problem that can be tackled by several approaches, such as grid search or Bayesian optimization [Shahriari et al., 2016]. The case of CNN brings several complications. First, the function evaluation becomes very expensive. Second, the number of hyperparameters is very large and may vary across the optimization space. Thus, hyperparameter optimization for CNN requires specialized approaches.

Perhaps the first successful attempt at automatic architecture search is a neural architecture search (NAS) algorithm [Zoph and Le, 2017] which utilizes reinforcement learning. The CNN architecture is predicted by a recurrent neural network that sequentially produces CNN hyperparameters: filer sizes, strides, numbers of filters. Possible branchings

and skip connections are modeled using an attention mechanism that decides which connection between the current layers and the previous layers are to be introduced in the next step.

NAS requires an extremely large amount of computational resources even when performed on small datasets. Thus in the experiments with the CIFAR-10 dataset, the authors report testing 12800 architectures during the search process and using 800 GPUs simultaneously. This makes architecture search for larger datasets such as ImageNet impossible.

Larger datasets can be approached in two ways: by designing more efficient search algorithms, or by ensuring that the results obtained on a small dataset are transferable to the larger datasets. Both approaches are implemented in [Zoph et al., 2017]. First, the modular structure is imposed on a target CNN. This way, only the architecture of a block has to be predicted instead of the whole network, making the search space much smaller. Second, CIFAR-10 and ImageNet versions of architectures can be made of the same blocks with a different number of poolings (or strided convolutions) between them. Architectures built this way beat human-constructed architectures on ImageNet.

The neural architecture search can be further accelerated by sharing parameters between different architectures [Pham et al., 2018, Cai et al., 2017] or by predicting the final performance of architecture based on the first epochs at the beginning of the training process [Baker et al., 2017]. With these enhancements, hundreds of GPUs are no longer necessary for automatic architecture search. Thus, [Cai et al., 2017] reports closely reproducing the original neural architecture search results using five GPUs instead of 800.

Reinforcement learning (RL) is not the only possible way to search in the space of CNN architectures. A genetic algorithm was carefully compared with the RL baseline in [Real et al., 2018]. They conclude that a genetic algorithm can match RL with quicker convergence.

Still, all the variations of approaches which divide architecture construction and evaluation require evaluating thousands of CNNs. The resource consumption can be drastically reduced if the architecture search and CNN training are done at the same time. MorphNets [Gordon et al., 2017] simultaneously learn CNN weights and change its architecture by iterating between the two stages. In the first stage, the network is thinned by the sparsity inducing regularizers. In the second stage, new channels are added uniformly to all layers. This algorithm allows CNN to adapt thickness of its layers to the particular task. Without the need to train large numbers of models, the training time of MorhNets is similar to regular CNN training, meaning the algorithm can be directly

applied on ImageNet. The downside of this approach is that the search space is limited to changing layer widths

2.4 Quantization

A switch to low-precision arithmetic or quantization is a straightforward way to speed up computations, as well as to compress and to minimize memory requirement for a neural network. This general idea produces a spectrum of approaches, which starts from using slightly lower precision and ends with the complete switch to binary weights and activation. While the binarization offers a compelling perspective of extremely low time and memory cost, it is not possible yet to fully transition to binary CNNs without substantial accuracy drops.

The problems faced by quantization and binarization are apparent. First, in the case of quantized weights, it is hard to implement gradient descent, since the idea of quantization contradicts the process of accumulating small changes. Second (and related), if the activations are quantized, the backpropagation process becomes complicated. This section surveys some of the ways to deal with these problems.

The list of approaches for improving the speed of neural networks on CPUs [Vanhoucke et al., 2011] includes 8-bit quantization. Several facts which facilitate quantization are listed. Firstly, because of the sigmoid activation function, activations stay in the $[0, 1]$ interval, so no scaling is needed. Secondly, because of the linear nature of the operation together with range compression by sigmoid, quantization errors tend to propagate sublinearly. That said, modern CNNs rarely use sigmoid activations, so this argument may no longer be valid, although sublinearity of error propagation still holds with ReLU activation. Moreover, [Hinton et al., 2012] notes that neural networks are not only robust towards the noise, but the training performance can be enhanced by noise injection. Thus, noise-like distortions injected by the quantization process may not be detrimental if this process is properly tuned. Therefore, if the hardware allows it, low precision arithmetic is a viable technique for speeding up neural networks.

The problem of the compression of fully-connected weights matrix W was addressed in [Gong et al., 2014]. The compression often goes hand-in-hand with the improvement of the speed, and some techniques are applicable both to convolutional and fully connected layers, so this work turns out to be very influential for speeding up CNNs. The following approaches are compared in [Gong et al., 2014]:

- **Truncated SVD-decomposition** approximates weights matrix $W \in \mathbb{R}^{n \times c}$ as a product of smaller matrices:

$$W \approx \hat{W} = USV^T = U'V^T \tag{2.2}$$

  where $U, U' \in \mathbb{R}^{n \times k}$ and $V \in \mathbb{R}^{c \times k}$. The input multiplication by W is then replaced by two multiplications with U' and V^T. The compression and speed-up rate are controlled by the number of components in the decomposition k.

- **Binarization,** which is the simplest and the most radical way to compress parameters by applying thresholding:

$$\hat{W}_{ij} = \begin{cases} 1 & W_{ij} > 0 \\ -1 & W_{ij} < 0 \end{cases} \tag{2.3}$$

  Binarization compresses data from full 32 bit precision to 1 bit, with the fixed compression rate of 32×.

- **Scalar quantization**: all entries of matrix W are clustered by the k-means algorithm, and the centroid values c_t are used for the approximation

$$\hat{W}_{ij} = c_t \quad \text{where} \quad t = \underset{z}{\operatorname{argmin}} |W_{ij} - c_z| \tag{2.4}$$

- **Product quantization** [Jegou et al., 2011] divides matrix W into several submatrices $[W_1, W_2 \dots W_s]$, then each submatrix is clustered by kmeans and compressed independently.

As opposed to the previous results for the convolutional layers, matrix decomposition performed poorly according to [Gong et al., 2014]. Simple scalar quantization and product quantization achieved much better results, and, surprisingly, the simplest binarization technique also worked reasonably well. Obviously, simplicity and high compression rate make binarization a very promising approach, but its capabilities are very limited unless one can properly train binarized networks and compensate the accuracy drop incurred by binarization. A multitude of attempts at reconciling binarization with backpropagation and gradient descent were done in the recent years.

The initial development of algorithms for neural network quantization was mostly done in the area of speech recognition, which is not covered in this thesis. For images, [Anwar et al., 2015] presents an algorithm for training quantized CNNs. The basic idea is to keep two versions of the weights: quantized $\hat{W}$ and full precision W. The algorithms repeat the following steps:

1. Obtain quantized weights with some sort of quantization procedure q applied to high-precision weights:
$$\hat{W} = q(W) \tag{2.5}$$

2. Perform a feed-forward pass with quantized weights and compute the loss function. The activations are kept in full precision.

3. Backpropagate the error gradients with quantized weights and full precision activations. The gradients are then used to update the full-precision weights.

This sequence allows circumventing the problems with backpropagation and gradient descent for quantized weights. Different quantization levels are used for different layers and the main benefit here is the model compression.

The presented results on MNIST and CIFAR10 datasets exceed uncompressed CNNs in some cases. This effect is attributed to the regularizing effect of quantization which reduces the CNN capacity (a similar effect was reported in [Hinton et al., 2012]). At the same time, there are other well-studied ways to reduce capacity, such as regularization, dropout, and simply reducing the number of filters inside CNN. The latter method also leads to speed-up and compression. Overall, the main disadvantage of [Anwar et al., 2015], shared with many others in the field, is that the experiments are limited to small networks with 32×32 inputs. Such CNNs have a relatively low capacity and are quick to experiment with. And yet, the main challenge of speeding up CNNs lies in the area of bigger CNNs (such as those trained for ImageNet classification and other similar tasks), to which good results on small images do not always transfer.

The **BinaryConnect** [Courbariaux et al., 2015] approach pushes the principle of splitting high-precision and quantized weights further to achieve full binarization, i.e. training of the convolutional networks with binary weights. The binarization procedure (2.3) is modified in a probabilistic fashion:

$$\hat{w} = \begin{cases} 1 & \text{with probability} \quad p = \sigma(w) \\ -1 & \text{with probability} \quad 1 - p \end{cases} \tag{2.6}$$

where σ is the hard version of sigmoid function

$$\sigma(x) = \text{clip}(\frac{x+1}{2}, 0, 1) \tag{2.7}$$

which is chosen because it is much less computationally expensive compared to the regular sigmoid function. The high-precision weights are clipped into $[-1; +1]$ interval during training. A probabilistic approach is in general desirable from the theoretical

point of view. On the other hand, the cost of random number generation accumulates if used on every step.

At train time, BinaryConnect repeats the following steps:

1. For every high-precision weight w, pick $\hat{w} = \pm 1$ according to (2.6).
2. Perform feed-forward pass with binarized weights $\hat{w}$.
3. Backpropagate with binarized weights $\hat{w}$ and update high-precision weights w.

Two ways of the model evaluation in test time are considered:

- Use the binarized weights $\hat{w}$. Forward propagation with binary weights can be much faster since it replaces floating-point multiplications by multiplications with ± 1, which is just a sign change.
- Use the real-valued weights w. This way, the binarization is only treated as a regularization technique, and no acceleration in test time is achieved.

Additionally, [Courbariaux et al., 2015] proposes generating binarized weights multiple times as test time to obtain the ensemble of models, but ensembling contradicts with the speed-up task and preserving full-precision weights contradicts compression. Competitive results are presented for CIFAR10, SVHN, and permutation-invariant MNIST, both for the real-valued and binarized weights. Again, as in the case with [Anwar et al., 2015], classification accuracy sometimes exceeds the full-precision baseline for small datasets.

Two extensions of this approach are presented in the follow-up paper [Lin et al., 2016]. First of all, ternary weights are introduced. Ternary weights are obtained by a stochastic procedure similar to (2.6). Every weight w is assumed to lie in the interval $[-1, 1]$. This interval is divided into two sub-intervals $[-1, 0]$ and $[0, 1]$ and the probability of picking 1, 0 or -1 is determined by the procedure (2.6) applied to the respective interval. The second important innovation is the elimination of multiplications in the backward pass. The layer activations are quantized into 3 or 4 bits and multiplication is replaced by bit-shifts.

The comparison of learning curves shows that binary and ternary networks behave similarly both with and without backward pass quantization: initially, convergence is slower, but the final result can be better. Again, this is attributed to the regularizing effect of quantization.

In a subsequent paper on binarized neural networks [Hubara et al., 2016a], the details on the practical implementation and timings are provided. Shift-based versions for

batch normalization [Ioffe and Szegedy, 2015] and ADAM optimization [Kingma and Ba, 2014] algorithm are presented. A custom CUDA kernel is written for binary matrix multiplication, and its speed is compared to cuBlas on 8192×8192 matrix multiplication. The binary kernel is reported to be $3.4\times$ faster. The number of possible binary filters is limited by the filter size. For example, with 3×3 filters there are $2^{3\times 3} = 512$ possible filters. This is a sign of extremely limited capability of binary networks but also an opportunity to save some computation time by applying unique filters only once. The preliminary result on the ILSVRC challenge with AlexNet architecture is 36.1% top-1 accuracy, which corresponds to $\sim$ 20% accuracy drop compared to the full-precision architecture.

Another interesting and more successful attempt at binarizing large CNNs is the XNOR-Net [Rastegari et al., 2016]. Here, the weight binarization setting is considered as an approximation problem of the following kind:

$$I * W \simeq \alpha(I * B)\,, \tag{2.8}$$

where $*$ denotes convolution, I is the input array, α is a scaling factor, W is the array containing high-precision weights and B is its binary version. It can be shown that the optimal values of elements of W are indeed obtained by the simple binarization procedure (2.3), and the optimal value for the scaling factor is an average of the absolute values of W. Training the binary weights network is done by repeating the same three steps from [Anwar et al., 2015, Courbariaux et al., 2015], i.e. obtaining binarized weights from high precision weights, doing forward and backward passes with binarized weights, and applying updates to the full precision weights.

The next step is to binarize both weights and activation, resulting in the so-called *XNOR-networks*. The approximation problem of the following form is considered:

$$I * W \simeq (\mathrm{sign}(I) * \mathrm{sign}(W)) \odot K\alpha\,, \tag{2.9}$$

where $\odot$ is an element-wise multiplication and K is the array with scaling factors for every patch in I. This approximation leads to $\times 58$ speed-up in terms of the number of the floating point operations, while the actual timings will depend on the cost of binary operations, which depends on the hardware and implementation details.

XNOR-Nets training also requires the following rearrangement of the traditional CNN block sequence, in order to minimize the information loss in binarization:

1. Batch normalizations are put at the beginning of the block.
2. Following batch normalization, the binary activation layer computes K and $\mathrm{sign}(I)$.

	full precision	binary weights	XNOR-Nets
AlexNet	56.6	56.8	44.2
ResNet-18	69.3	60.8	51.2

TABLE 2.2: ImageNet (ILSVRC) classification accuracy of binarized CNNs from [Rastegari et al., 2016]. The accuracy drop is large compared to tensor decomposition methods, but the speed-up and compression rates assosiated with binarization are much higher.

3. Binary convolution is applied to the result of the binary activation layer.

4. Optionally, pooling is applied.

The blocks of layers composed in the same way are then applied several times.

Overall, the accuracy of binarized XNOR-networks is shown in the table 2.2. Interestingly, the accuracy for full-precision AlexNet and its version with binarized weights is the same, although this effect does not hold for larger architectures. Thus for ResNet-18, XNOR-Net loses more than 10% of accuracy compared to the full-precision network.

Generally speaking, binarization is an approach with low flexibility: it promises extremely large speedups, but often incurs substantial accuracy drop which may be unacceptable in practical applications. One way to cover this gap is to dial compression rate back and return from binarization to low-bit quantization. Towards this end, quantization with different compression rates is considered in [Hubara et al., 2016b]. A quantized version of AlexNet with 1-bit weights and 2-bit activations achieves 51% accuracy. Varying quantization levels for weights, activations, and gradients are tried in [Zhou et al., 2016]. Another way to boost the accuracy of binarized CNN is by increasing the number of channels. This approach was found beneficial in [Mishra et al., 2017]

Yet another quantization based approach with more flexibility is Lookup-based CNN (LCNN) introduced in [Bagherinezhad et al., 2017]. Interestingly, LCNN utilizes ideas of decomposition, quantization, and sparsity at the same time. First of all, the convolutional weights W are decomposed into the sum of vectors of the dictionary matrix D, with coefficients C and indices I:

$$W(i,j,:,t) = \sum_{\xi=1}^{s} C(\xi,i,j)D(I(\xi,i,j),:) \tag{2.10}$$

The two spatial dimensions and the last dimension corresponding to the output channel index are not affected by the decomposition. Following this insight, one may take advantage of the fact that the general convolution can be expressed trough 1×1 convolutions

	AlexNet		ResNet-18	
	accuracy	speedup	accuracy	speedup
CNN	56.6	1×	69.3	1×
LCNN-accurate	55.4	3.2×	62.2	5×
LCNN-fast	44.3	37.6×	51.8	29.2×

TABLE 2.3: LCNN accuracy on ImageNet (ILSVRC) classification task. The performance of LCNN can be tuned by changing dictionary size and the number of components in the decomposition. Two variants of the algorithm are shown in this table. The speed-ups are measured in terms of FLOPs, and the actual "wall-clock" speed-ups are likely to be much lower on most architectures.

and shift operations:

$$V(x,y,t) = \sum_{i=0}^{d}\sum_{j=0}^{d}\sum_{s=1}^{S} W(i,j,s,t)[\underset{ij}{\text{shift}}\, U](x,y,s)\,, \tag{2.11}$$

where $\text{shift}_{i,j}$ indicates the spatial shift operation. Combining (2.10) and (2.11) yields the following way to perform generalized convolutions:

$$V(x,y,t) = \sum_{i=0,j=0}^{d,d} \text{shift}_{i,j} \sum_{\xi=1}^{s} C(\xi,i,j)S \tag{2.12}$$

where the array S contains the result of 1×1 convolutions of the input U with the filters from D. Shifts, scaling and 1×1 convolution, which is implemented through matrix multiplication, are all relatively inexpensive multiplication. The cost of this pipeline can be regulated by changing the dictionary size.

Direct training of the proposed lookup based convolution is a combinatorial optimization problem. To get around this complication, the lookup and scale stage are reformulated using a standard convolution with sparsity constraints. Reported speedups of LCNN reach 37.6×, as shown in the table 2.3. However, this value refers to the number of floating point operations, which may not translate well to actual timings, especially for the architecture that heavily relies on the lookups (which are known to be relatively slow on most architectures).

To summarize, weight quantization or binarization is an effective technique for CNN compression. As for the speed-up, the published works paint a mixed picture. It is clear that quantization of CNN weights or activations allows for faster computations, but the actual speedup depends on a particular low-level implementation. Most of the time researchers do not publish such implementations, and when they do, it appears that existing implementations of floating point operations are very well optimized and the actual speedups brought by quantization methods are not nearly as high as the operation-count based prediction suggests.

2.5 Pruning

Pruning away parts of the convolutional weights is a natural way to reduce the complexity of the convolutional operation. This approach, applied for speeding up convolutions in neural networks, is a popular research topic with a very large number of publications. Starting from the optimal brain damage [LeCun et al., 1990], this is perhaps the oldest approach among listed in this review.

Most of the pruning approaches follow the same pipeline. Starting from the pretrained baseline, the following two steps are applied, possibly iteratively. First, the importance of neurons is calculated according to some criterion. The least important neurons are pruned. Then, the network is fine-tuned leading to partial recovery of the accuracy drop. In the case of the iterative process, sparsity-inducing regularizer may be applied during the fine-tuning stage.

Three basic choices have to be made to implement this pipeline. First, the desired sparsity structure must be chosen. Second, the importance (pruning) criterion should be selected. Finally, a sparsity-inducing regularizer should be chosen (if the approach uses one). Below, different design choices along these three axes are reviewed.

Sparsity structure. Pruning individual weights does not necessarily results in a speed-up. Assume the convolution is implemented through `im2col` and matrix multiplication, as described by [Chellapilla et al., 2006]. In this implementation, most of the computation time is spent inside the matrix multiplication

$$\hat{V} = \hat{W}\hat{U} \tag{2.13}$$

where $\hat{V}$ is the output in the matrix-reshaped form, $\hat{W}$ is the convolutional weights array also reshaped as a matrix and $\hat{U} = \text{im2col}(U)$ is the patch matrix obtaining by copying and rearranging of input array U by the im2col operation. The columns of the patch matrix correspond to input patches of size $d_x \times d_y \times S$.

As some elements of W will be replaced by zeros by the pruning algorithm, one can switch to some sparse representation for W. In the lack of the structure of the sparsity, sparse matrix multiplication carries significant overhead compared to dense matrix multiplication. As shown in Figure 2.4, sparse version becomes faster only if the density of W is well below 0.1, which is unreachable in the practical setting without a significant drop in accuracy. The only way to overcome this problem is to arrange elements of W into groups and to use structured sparsity. While the use of `im2col` is not the only way to implement generalized convolution, other implementations follow similar patterns and likewise cannot benefit from medium levels of unstructured sparsity.

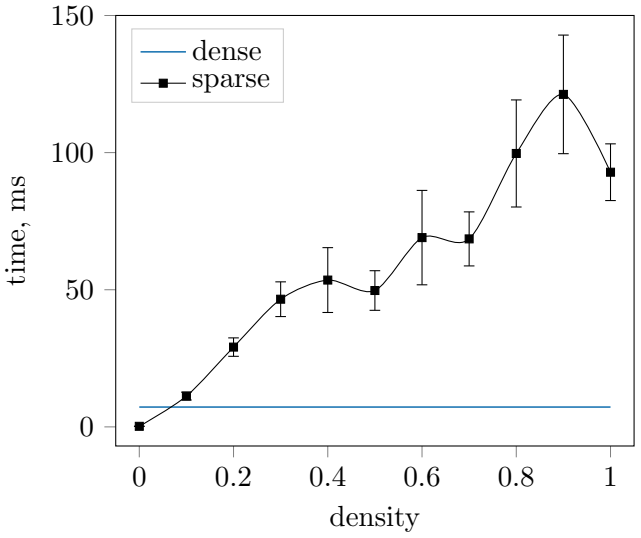


FIGURE 2.4: The comparison of matrix multiplication with sparse (CSC representation) and dense weight matrices. Sparse version becomes more efficient only for densities smaller than 0.1. These CPU (Intel Core i7-6600U) timings are computed via numpy and scipy.sparse libraries for matrices with sizes corresponding to 3×3 convolutions with 64 channels and 64×64 maps. The elements of the matrices are sampled from the uniform distribution over $[0, 1)$.

The consideration discussed above calls for the use of structured sparsity during pruning. The finest possible division into groups is considered in [Lebedev and Lempitsky, 2016] and [Liu et al., 2015a]. Their algorithms remove columns from the weight matrix $\hat{W}$ and corresponding rows from patch matrix $\hat{U}$. The removal is facilitated by the custom version of the `im2col` function which omits elements corresponding to deleted parts of $\hat{W}$ while constructing the patch matrix. With this kind of structured sparsity, the original matrix multiplication is replaced with the multiplication of smaller matrices, which are still dense. This leads to the speedups that are almost directly proportional to density.

A related but orthogonal approach to structured sparsity called *perforation* was proposed in [Figurnov et al., 2016]. The main idea is to remove columns from the patch matrix $\hat{U}$. Since columns correspond to image patches, this means the convolution will not be computed for some subsets of points in the image. The output value in these points can be interpolated or effectively omitted if the next layer performs the pooling operation.

Several additional ways to organize sparsity are proposed in [Wen et al., 2016] and [Shin et al., 2018]. Each of them is defined by the specific way of slicing the four-dimensional weight array W:

- Slicing in the form of $W(i, j, s, :)$ is the same as in the group-wise sparsity approach and produces non-square filters. It is the finest division that can be implemented efficiently, but it requires specialized implementation.

- Removing $W(i, j, :, :)$ cuts all filters in the layer simultaneously. This slicing can be used to trim the filter size, for example from 5×5 to 3×3.
- A whole filter corresponds to slice $W(:, :, s, :)$. With such slice set to zero, the s-th output channel will be filled with zeros. The complexity of the network then can be decreased by removing slices from the weight arrays of the current and the next convolutional layers.
- Removing $W(:, :, :, t)$ cuts all the connection with t-th input channel, which means this channel can be removed.
- Removing $W(:, :, s, t)$ cuts all the ties between the s-th input and the t-th output channels. This slicing can be used to turn full convolution into group convolution [Krizhevsky et al., 2012].
- Finally, in the case of residual architecture, the convolutional weights W can be set to zeros completely. This operation removes the whole residual block of the network.

Pruning criteria. Here, we follow the notation of [Molchanov et al., 2016], which contains a similar review of criteria, and denote the pruning criterion by Θ. The simplest criterion is the **absolute value of the weight**:

$$\Theta(w) = |w| \tag{2.14}$$

It was successfully used in [Han et al., 2015] for pruning individual weights, and then in [Lebedev et al., 2015] for groups. This criterion is consistent in the sense that if the weight already equals zero it can be safely pruned, but small non-zero weight can be disproportionally important if it acts on a large activation or pushes some points across the decision surface.

Another choice is to focus not on the weights, but on the **activations** a:

$$\Theta(a) = \sum_i a_i^2 \tag{2.15}$$

Since ReLU non-linearity naturally produces sparse activations, there is a realistic chance to find groups of neurons which can be safely pruned. The average percentage of zeros is a different metric proposed in [Hu et al., 2016] for this situation.

Mutual information measures the dependence of two random variables. In theory, mutual information between activation group and targets $I(a, y)$ would be an excellent pruning criterion, but the direct computation is too complex, and available approximations are not performing well according to [Molchanov et al., 2016].

Taylor expansion of the objective function can be used to estimate its change after the perturbation caused by the pruning process. For example, the original Optimal Brain Damage paper [LeCun et al., 1990] used the criterion based on the second-order Taylor decomposition. Assuming that C is the learning objective, the approach makes an assumption that $\frac{\partial C}{\partial w_i} = 0$ ("extremal approximation"), which holds when the learning has fully converged. Assuming that the mixed derivatives could be neglected, the approach then uses the non-mixed second-order derivatives as the pruning criterion:

$$\Theta(w_i) = \frac{1}{2}\frac{\partial^2 C}{\partial w_i^2} w_i \tag{2.16}$$

The necessity to compute second derivatives made this approach unpopular, since this capability was not implemented in the deep learning frameworks until recent years. The works [Molchanov et al., 2016, Figurnov et al., 2016] avoid the extremal approximation and use the following criterion:

$$\Theta(a_i) = \left| \frac{\partial C}{\partial a_i} a_i \right| \tag{2.17}$$

This criterion is expressed through the values which can be computed by the standard back-propagation process. In general, a comparison of pruning criteria listed above performed by [Molchanov et al., 2016] demonstrated the superiority of the Taylor-expansion based criteria.

The ThiNet approach [Luo et al., 2017] focuses on pruning filters and proposes a special criterion for this case. The key observation is that if the filter is pruned from the i-th layer, the corresponding output channel will be empty, and the same channel should be pruned in the kernel of the $(i+1)$-th layer. The next, $(i+2)$-th layer will then be the first subsequent layer, whose input data size is not affected by the change. Thus, a natural pruning criterion relies on the reconstruction error of the inputs to the $(i+2)$-th layer. After the pruning, the channel scaling computed via least squares can be used to reduce the error, although this step cannot replace the fine-tuning.

Regularizer. The pruning process can work without sparsity-inducing regularization, but the sparsity-inducing regularization can help the pruning process while incurring a minimal computational overhead. The $L1$ regularizer $\Omega_1(w) = \lambda|w|$ induces unstructured sparsity, but for structured sparsity, $L2,1$ regularization can be used:

$$\Omega_{2,1}(w) = \lambda \sum_i \sqrt{\sum_{j \in g_i} w_i} \tag{2.18}$$

Here, g_i are the weight groups, defined by one of the ways described above. A smart approach for achieving filter-level sparsity was proposed in [Liu et al., 2017]. They notice that in modern CNNs, convolutions are almost always followed by batch normalization.

The filter level sparsity can then be achieved simply by the L_1 regularization imposed on the scaling factors within batch normalization.

2.6 Teacher-student approaches

Teacher-student approaches follow the idea that a CNN model can be trained on the outputs of another model (a teacher), as opposed to regular training on labeled data. This approach allows to transfer knowledge from one model to another and to incorporate unlabelled or synthetic data into the training process (as an unlabeled example can still be passed through the pretrained teacher model). Originally, such transfer was performed from a non-interpretable model such as a neural network to more interpretable ones, such as decision trees [Craven and Shavlik, 1996], or a set of rules [Thrun, 1995]. Another natural purpose for the teacher-student approach would be to transfer knowledge from large, slow and accurate models to small and fast ones.

Towards this end, [Bucila et al., 2006] propose to compress an ensemble of models into a single neural network. First, an ensemble of classification models is trained on a certain annotated dataset. An ensemble is expected to be more resistant to overfitting compared to a single model. This ensemble is used to label a large amount of synthetic data, generated by several simple random sampling procedures, and finally, a single model is learned on the resulting synthetic dataset. [Bucila et al., 2006] state that this approach can alleviate the overfitting problem for neural networks without time and memory costs of building an ensemble. It should be noted, though, that this work was done on small datasets with fully-connected neural nets, and utilized data generation methods that are not directly applicable to images. With modern CNNs, the viability of this approach is limited by the following facts: datasets are already very large and the models are too large to build large ensembles.

A specific way of representing labels for synthetic data is a key detail of knowledge transfer algorithm. The description of same ensemble compression idea in [Zeng and Martinez, 2000] elaborates on the importance of preserving not just the labels, but whole vectors of a posterior probability distribution over the output classes. Such vectors capture richer information about the actual content of data samples, making knowledge transfer process more efficient.

This idea of utilizing full probability distribution is further expanded in [Hinton et al., 2014]. The proposed distillation procedure requires training a student model on the soft version of the outputs of the original (teacher) model. Let z_i be the raw outputs of the neural network, and p_i be the output probabilities. These probabilities are then

calculated according to the softmax formula:

$$p_i = \frac{\exp z_i/\tau}{\sum_j \exp z_j/\tau} z\,, \tag{2.19}$$

where τ is the temperature parameter. Let p^S be the probabilities for the student model, and p^T be the probabilities of original teacher model. The student model is trained to approximate both the correct labels y and the outputs of teacher model using the following loss function:

$$L = H(p^S, p^y) + \lambda H(p^S, p^T)\,, \tag{2.20}$$

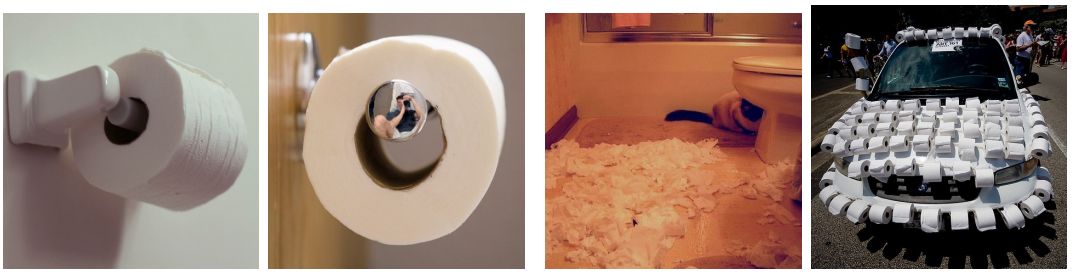
where H is the cross-entropy and p^y is the one-hot distribution corresponding to the ground truth. A high temperature τ effectively regularizes the student model, while the lower temperature allows transferring knowledge in finer detail. In practice, the temperature parameter τ has to be tuned manually. It can be shown that in the limit of high temperature this procedure is equivalent to training on raw outputs z_i (the regime which was utilized in [Li et al., 2014] for acoustic model compression). Results on MNIST, an automatic speech recognition task and large-scale image classification task are presented, and a significant rate of model compression is achieved for all tasks. Most impressively, it is shown that knowledge transfer can be successful even if one of the classes is missing from the dataset used for the transfer since the information about this class is still carried through soft labels of other classes.

The idea of distillation was extended to multiple layers of deep networks in the FitNets approach [Romero et al., 2015]. They introduce the notion of a *hint*, which is defined as the output u_T of the teacher's hidden layer, and connect thus layer with a hidden layer of the student CNN, which they call the *guided layer*. The guidance process is implemented via the addition of the Euclidean distance between the hint and the guided layer output u_S to the loss function (2.20). When the layer sizes of the hint and the guided layer differ, the linear regressor r that maps the hints to the guided layer activations is added into the training leading to the following term (the *guidance loss*):

$$L_h = \frac{1}{2}\|u_T - r(u_S)\|^2 \tag{2.21}$$

The fact that the student now not only has access to the outputs of the teacher but also receives insights from the internal data representation, leads to faster convergence and better performance of the method.

Teacher-student approach is a powerful tool which can be used to help training of



(A) easy (B) confusing

FIGURE 2.5: Samples from ILSVRC validation set toilet paper class. Some samples are easy to classify, while others are confusing. Powerful model capable of producing the correct labels for the samples on the right is an overkill for the samples on the left.

quantized networks. It has been successfully applied to the training of quantized networks [Polino et al., 2018] and ternary networks [Alemdar et al., 2017].

In the modern era of neural networks, most papers focus on the situation where both the teacher and the student are neural networks. While it is logical to use the best method available as a teacher, the need for the faster student may lead to different kinds of models. Thus, [Frosst and Hinton, 2017] proposes to use soft decision tree as a student model. Decision tree, in theory, can provide very high speed-ups, but in this paper only results on MNIST and Connect4 datasets are presented, while the achieved MNIST accuracy of 96.76% is below modern standards.

2.7 Adaptive methods

Pushing to the limit the idea of automatic architecture tuning, we get the model which processes samples from the same dataset differently. This kind of model may adapt to the complexity of the sample, processing easy samples quickly and spending more time on the complex ones. An example of complexity range in the ILSVRC dataset is shown in Figure 2.5.

Variable complexity is natural for object detection algorithms with proposal generation step: the bigger is the time budget, the more proposals should be generated and evaluated. For image classification and other tasks solved with a single CNN, network structure has to be changed to change inference time.

A natural way to achieve variable inference time is through variable depth, which can be easily done ResNet-type architecture by removing some of its blocks. This was first proposed and explored by [Huang et al., 2016], as a measure to reduce training time for extremely deep models. Thus, in the training time network has varying depth during training and large fixed depth during test stage.

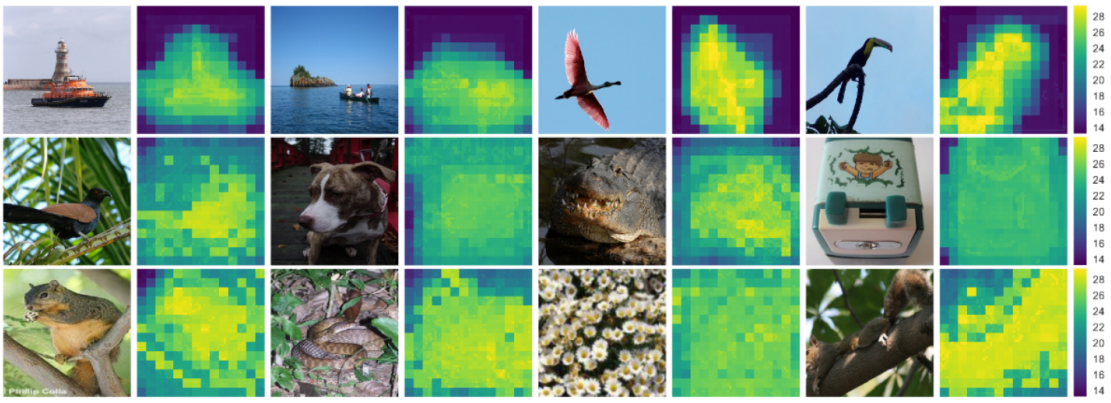


FIGURE 2.6: Ponder cost maps for images from ILSVRC validation. Higher ponder cost corresponds to longer computation by SACT algorithm [Figurnov et al., 2017]. Large areas of low ponder cost associated with uniform image areas demostrate the capability of SACT to reduce redundancy on these images. Top: low ponder cost (19.8-20.55), middle: average ponder cost (23.4-23.6), bottom: high ponder cost (24.9-26.0).

The architecture that changes depth adaptively, called Adaptive Computation Time (ACT), was proposed by [Graves, 2016] for recurrent neural networks. A special branch, added to each layer, predicts a halting score h, a single value lying in the $[0, 1]$ range. These halting score can be interpreted in the probabilistic or deterministic fashion. In the first case, halting is simply a probability to stop at the current layer. In the second case, the computation stops then the sum of halting scores exceeds 1. To avoid the unlimited growth of computation time, a special loss term called *ponder cost* which penalizes low halting scores is introduced. Maps of ponder cost demonstrate what SACT provides an attention mechanism, as shown in Figure 2.6.

The ACT was extended for the image processing as Spatially Adaptive Computation Time(SACT) by [Figurnov et al., 2017]. SACT applies ACT mechanism for every spatial location, i.e. halting score computation and decision to finish is done separately for each location. This means that computation will continue for some positions when it is finished for others. Such a process can be implemented through perforated convolution [Figurnov et al., 2016], which is described in the Section 2.5.

The approaches listed above regulate depth by halting, which is a logical way to address the difference between easy and complicated samples of the same nature. In other cases, it may be desirable to have some specialization on higher layers of the network. [Wang et al., 2017] proposes to use dynamically routed networks (SkipNets) instead of ACT. Training SkipNets is challenging, as skipping some layers in the middle of data flow is an inherently discrete decision, and the attempts to train it with some sort of soft approximations lead to substantial accuracy drop at test time when hard thresholding have to be introduced. As a result, the complex learning algorithm which uses soft approximations and reinforcement learning during different stages is required.

SkipNets outperform ACT and SACT for classification tasks, although the final results on ImageNet (up to 30% speedup for ResNet-101 and only 12% for ResNet-50) are not very impressive.

Complex learning procedures are avoided by [Bolukbasi et al., 2017], who builds a model from pretrained CNNs. In the case of ImageNet, three networks are used: AlexNet, GoogleNet, and ResNet-50. The combined model evaluates the AlexNet first and then makes a decision to adopt AlexNet's answer or go to one of the more complex models, or to the both of them. Resulting adaptive model shows up to a 2.8× speedup then compared to ResNet-50 with 1% loss of top5 accuracy, although the results are worse for top1 accuracy. By comparison with efficient architectures from Section 2.2 and Section 2.3, this approach is limited in the sense that it can not be both faster and more accurate than any of the used preexisting CNNs.

[Teerapittayanon et al., 2016] propose BranchyNet, a single neural network having several side-branches with jointly trained classifiers. BranchyNet utilizes its lower branches to allow the samples to exit early, thus reducing the cost of inference. The entropy of an output distribution is used as a measure of confidence in the prediction. Then the confidence goes over the threshold, the computation stops. Tuning the threshold value allows controlling speed-accuracy tradeoff without retraining the model.

2.8 Problem-specific approaches

Most of the approaches listed above are developed and compared on classification problems. Deep learning in computer vision, however, is not limited to classification. In recent years deep learning has been successfully applied to a wide variety of tasks, and for some of them time constraints are critical and very specific approaches can be applied to satisfy these constraints.

Probably the most notable example of progress in fast problem-specific approaches in computer vision is an object detection.

Regions with CNN (R-CNN). R-CNN[Girshick et al., 2014] works in two stages. On the first stage, selective search algorithm produces an excessive number of proposals. Then, CNN accesses each proposal and classifies it them as a false positive or one of the object classes. R-CNN is slow because it runs image patches through CNN multiple (thousands) times. R-CNN can be naively accelerated by accelerating CNN with any of the approaches listed in the previous sections, but the more advantageous choice is to redesign the core algorithm to minimize the number of forward passes.

R-CNN runs a neural network on a lot of intersecting patches independently. This source of redundancy was realized and exploited by [Girshick, 2015]. Faster R-CNN shares all the feature extraction steps: the fully-convolutional neural network is run only once, it produces output data with some spacial resolution, and the final descriptor of the region is obtained by pooling. Faster-RCNN does not run convolutional networks multiple times but still requires a separate step to generate proposals and evaluates them separately.

The next generation of object detection algorithms, such as Faster R-CNN[Ren et al., 2015] and YOLO[Redmon et al., 2016, Redmon and Farhadi, 2016, 2018], completely abolishes region proposal step. Instead, the regions are proposed by the same single network which computes features on the whole image.

To summarize, the object detection started with limited usage of CNNs inside the larger complex pipeline and came toward large CNN trained for multiple objectives.

Similar trends can be observed in the area of texture generation and image stylization. An initial introduction of CNNs as feature extractors to this area by [Gatys et al., 2015], who introduces an image style similarity function f computed on top of features extracted from the VGG network. The problem of texture generation is then reduced to the minimization of f with respect to pixel values of the generated image. Minimization problem is solved by an iterative algorithm, which requires passing data through VGG network on every iteration. Again, speeding up this network would never be as fruitful as redesigning the algorithm, which was done by [Ulyanov et al., 2016] and [Johnson et al., 2016]. They proposed to train a separate image generator network, using f to write down its loss function. Then trained, this network generates samples of the same quality as an iterative algorithm, but thousands of times faster.

2.9 Summary

Some approaches, such as tensor decomposition based methods, seem to reach saturation, as many decompositions have been tried and it hard to come up with a novel idea. With other approaches, e.g. those based on binarization, there is a lot of work on algorithms and implementation that can be done in the future. Here, transferring impressive speed-ups in terms of the number of operations into actual wall clock speed-ups remains a challenge.

Designing efficient architectures is probably the most practical approach, as it does not require complex multi-stage processes that interleave modifications and finetuning stages. While it seems that in the future these architectures will be constructed automatically, some basic modules or design ideas are likely to come from humans rather

than from automated search. Still, while automatic architecture search is a young area of research, it has already made an impact and it is safe to assume that it will continue to grow in importance in the nearest future.

There is probably a reasonable space for the search of the optimal combination of approaches from several groups, notably from those that speed-up existing architectures (tensor decomposition, quantization, pruning, teacher-student approaches). While these groups are not "orthogonal" as they exploit similar kind of redundancy in the original architecture, there may still be considerable benefits in combining approaches from different groups. Automated discovery of optimal mix-and-match combinations may be promising.

Despite the large body of work on the topic, the area of research concerned with speeding up CNNs (as well as designing efficient architectures "from scratch") is far from saturation, and we firmly believe that significant improvements can and will be made in the nearest future.

Chapter 3

CP-decomposition of convolutional weights

A group of works reviewed in Section 2.1 have achieved significant speed-ups of CNN convolutions by applying various tensor decompositions to the weights and activations of convolutional layers. In this chapter, we focus on the low-rank CP-decomposition, and apply it to the weights of the convolutional layer, to obtain a sequence of four new convolutional layers with small filters. These layers work faster, have fewer parameters and use only existing CNN building blocks. Once a convolutional layer is approximated and replaced, it is straight-forward to fine-tune the entire network on training data using back-propagation.

The CNNs obtained with the proposed method are somewhat surprisingly accurate, given that the number of parameters is reduced several times compared to the initial networks. Practically, this reduction in the number of parameters means more compact networks with reduced memory footprint, which can be important for architectures with limited RAM or storage memory. Such memory savings can be especially valuable for feed-forward networks that include convolutions with spatially-varying filters (locally-connected layers).

On the theoretical side, these results confirm the intuition that modern CNNs are over-parameterized, i.e. that the sheer number of parameters in the modern CNNs are not needed to store the information about the classification task but, rather, serve to facilitate convergence to good local minima of the loss function.

In the following section, we review the concept of CP-decomposition, revisit two of related works [Denton et al., 2014, Jaderberg et al., 2014b] most similar to one described in this chapter, and move on to describe the proposed method.

3.1 Method

3.1.1 Related works

Using low-rank decomposition to accelerate convolution was suggested by [Rigamonti et al., 2013] in the context of codebook learning, and then applied to CNNs by [Jaderberg et al., 2014b]. A decomposition suggested by [Jaderberg et al., 2014b] Figure 3.1b effectively approximates the 4D array of weights as a composition (product) of two 3D tensors (below, we call it two-component decomposition).

Once the decomposition is computed, [Jaderberg et al., 2014b] perform local fine-tuning with L2 loss on the deviation between the full and the approximated convolutions outputs on the training data. Differently from [Jaderberg et al., 2014b], our method fine-tunes the entire network based on the original discriminative criterion. While [Jaderberg et al., 2014b] reported that such discriminative fine-tuning was inefficient for their scheme, we found that in our case it works well, even when CP-decomposition has large approximation error. Below, we provide a theoretical complexity comparison and empirical comparison of our scheme with [Jaderberg et al., 2014b].

In the work that is most related to ours, [Denton et al., 2014] have suggested a scheme based on the CP-decomposition of parts of the convolutional weights obtained by biclustering (alongside with different decompositions for the first convolutional layer and the fully-connected layers). Biclustering of [Denton et al., 2014] separately splits the two non-spatial dimensions into subgroups and uses resulting grouping to cut weight tensor into a bunch of smaller tensors to reduce the effective ranks in the CP-decomposition. CP-decompositions of the convolutional weights in [Denton et al., 2014] have been computed greedily, increasing the approximation rank by one at each step without changing the approximation obtained on the previous step. Although the decomposition allows separating all four dimensions of the tensor, Denton et al. [2014] abandons the separation of the spatial dimensions, mentioning that the resulting gain was minimal. In this chapter we essentially simplify the approach of [Denton et al., 2014] by not performing biclustering and applying CP-decomposition directly to the full tensor. The greedy computation of CP-decomposition is also replaced with non-linear least squares method. Finally, we use all four components of the decomposition and, as discussed above, we finetune the whole network with backpropagation, whereas [Denton et al., 2014] only fine-tunes the layers above the approximated one.

3.1.2 CP-decomposition

Tensor decompositions are a natural way to generalize low-rank approach to a multidimensional case. Recall that a low-rank decomposition of a matrix A of size $n \times m$ with rank R is given by:

$$A(i,j) = \sum_{r=1}^{R} A_1(i,r)A_2(j,r), \quad i = \overline{1,n}, \quad j = \overline{1,m}, \tag{3.1}$$

and leads to the idea of separation of variables. The most straightforward way to separate variables in case of many dimensions is to use the canonical polyadic decomposition (CP-decomposition, also called as CANDECOMP/PARAFAC model). This decomposition, first proposed by Hitchcock [1927], was later rediscovered multiple times (see [Kolda and Bader, 2009] for a detailed review). For a d-dimensional array A of size $n_1 \times \cdots \times n_d$ a CP-decomposition has the following form

$$A(i_1, \ldots, i_d) = \sum_{r=1}^{R} A_1(i_1, r) \ldots A_d(i_d, r), \tag{3.2}$$

where the minimal possible R is called the canonical rank. The profit of this decomposition is that only $(n_1 + \cdots + n_d)R$ elements have to be stored instead of the whole tensor with $n_1 ... n_d$ elements.

In two dimensions, the low-rank approximation can be computed in a stable way by using singular value decomposition (SVD) or, if the matrix is large, by rank-revealing algorithms. Unfortunately, this is not the case for the CP-decomposition when $d > 2$, as there is no finite algorithm for determining the canonical rank of a tensor [Kolda and Bader, 2009]. Therefore, most algorithms approximate a tensor with different values of R until the approximation error becomes small enough. This leads to the point that for finding good low-rank CP-decomposition certain tricks have to be used. A detailed survey of methods for computing low-rank CP-decompositions can be found in [Tomasi and Bro, 2006]. `Tensorlab` [Sorber et al., 2014] software package was used to calculate CP-decomposition. In the variety of available optimization techniques, we chose the non-linear least squares (NLS) method. This method minimizes the Frobenius norm of the approximation residual (for a user-defined fixed R) using Gauss-Newton optimization [Phan et al., 2013].

Such NLS optimization is capable of obtaining much better approximations than the strategy of greedily finding the best rank-1 approximation of the residual vectors used in [Denton et al., 2014]. The fact that the greedy rank-1 algorithm may increase tensor rank can be found in [Stegeman and Comon, 2010, Kofidis and Regalia, 2002]. A simple

example highlighting this advantage of the NLS is also presented in appendix to Lebedev et al. [2015].

3.1.3 Convolutional weights approximation

The generalized convolution that maps an input array $U(\cdot,\cdot,\cdot)$ of size $X\times Y\times C$ into an output array $V(\cdot,\cdot,\cdot)$ of size $(X-d+1)\times(Y-d+1)\times N$ using the following linear mapping:

$$V(x,y,k) = \sum_{i=1}^{d}\sum_{j=1}^{d}\sum_{c=1}^{C} W(i,j,c,k)\,U(x+i,y+j,c)\,, \tag{3.3}$$

Here, $W(\cdot,\cdot,\cdot,\cdot)$ is a 4D array of convolutional weights of size $d\times d\times C\times N$ with the first two dimensions corresponding to the spatial dimensions, the third dimension corresponding to different input channels, the fourth dimension corresponding to different output channels. The spatial width and height of the filters are denoted as d.

The rank-R CP-decomposition (3.2) of the 4D convolutional weights has the form:

$$W(i,j,c,k) = \sum_{r=1}^{R} W^x(i,r)\,W^y(j,r)\,W^c(c,r)\,W^k(k,r)\,, \tag{3.4}$$

where $W^x(\cdot,\cdot)$, $W^y(\cdot,\cdot)$, $W^c(\cdot,\cdot)$, $W^k(\cdot,\cdot)$ are the four components of the composition representing 2D tensors (matrices) of sizes $d\times R$, $d\times R$, $C\times R$, and $N\times R$ respectively.

Substituting (3.4) into (3.3) and performing permutation and grouping of summands gives the following expression for the approximate evaluation of the convolution (3.3):

$$V(x,y,k) = \sum_{r=1}^{R} W^k(k,r)\left(\sum_{i=1}^{d} W^x(i,r)\left(\sum_{j=1}^{d} W^y(j,r)\left(\sum_{c=1}^{C} W^c(c,r)U(x+i,y+j,c)\right)\right)\right) \tag{3.5}$$

Based on (3.5), the output array $V(\cdot,\cdot,\cdot)$ can be computed from the inputs $U(\cdot,\cdot,\cdot)$ via a sequence of four convolutions with smaller filters (Figure 3.1):

$$U^c(x,y,r) = \sum_{c=1}^{C} W^c(c,r)\, U(x,y,c) \tag{3.6}$$

$$U^{cy}(x,y,r) = \sum_{j=1}^{d} W^y(j,r)\, U^s(i,y+j,r) \tag{3.7}$$

$$U^{cyx}(x,y,r) = \sum_{i=1}^{d} W^x(i,r)\, U^{cy}(x+i,y,r) \tag{3.8}$$

$$V(x,y,k) = \sum_{r=1}^{R} W^k(k,r)\, U^{cyx}(x,y,r)\,, \tag{3.9}$$

where $U^s(\cdot,\cdot,\cdot)$, $U^{cy}(\cdot,\cdot,\cdot)$, and $U^{cyx}(\cdot,\cdot,\cdot)$ are intermediate tensors (map stacks).

3.1.4 Implementation and Fine-tuning

Computing $U^c(\cdot,\cdot,\cdot)$ from $U(\cdot,\cdot,\cdot)$ in (3.6) as well as $V(\cdot,\cdot,\cdot)$ from $U^{cyx}(\cdot,\cdot,\cdot)$ in (3.9) represents 1×1 convolutions that essentially perform pixel-wise linear re-combination of input maps. Computing $U^{cy}(\cdot,\cdot,\cdot)$ from $U^c(\cdot,\cdot,\cdot)$ and $U^{cyx}(\cdot,\cdot,\cdot)$ from $U^{cy}(\cdot,\cdot,\cdot)$ in (3.7) and (3.8) are "standard" convolutions with small filterss that are "flat" in one of the two spatial dimensions.

In this chapter, we use `Caffe` [Jia et al., 2014b] framework to implement the resulting architecture, utilizing standard convolution layers for (3.7) and (3.8), and an optimized 1×1 convolution layers for (3.6) and (3.9). The resulting architecture is fine-tuned through standard backpropagation (with momentum) on training data. We can fine-tune all network layers including layers above the approximated layer, layers below the approximated layer, and the four inserted convolutional layers. However, the gradients within the inserted layers are prone to gradient explosion, so one should either be careful to keep the learning rate low, or fix the weights in some or all of the inserted layers, while still fine-tuning layers above and below.

3.1.5 Complexity analysis

Initial convolution operation is defined by NCd^2 parameters (number of elements in the tensor) and requires the same number of "multiplication+addition" operations per pixel.

For [Jaderberg et al., 2014b] this number changes to $Rd(C+N)$, where R is the rank of the decomposition (see Figure 3.1 and [Jaderberg et al., 2014b]). While the two numbers are not directly comparable, assuming that the required rank is comparable or several times smaller than C and N (e.g. taking $R \approx \frac{CN}{C+N}$), the scheme [Jaderberg et al., 2014b] gives a reduction in the order of d times compared to the initial convolution.

For [Denton et al., 2014] in the absence of bi-clustering, the complexity is $R(C+d^2+N)$, and for our approach it is $R(C+2d+N)$ (again, both for the number of parameters and for the number of "multiplications+additions" per output pixel). Almost always, $d \ll T$, which for the same rank gives a further factor of d improvement in complexity over [Jaderberg et al., 2014b] (and an order of d^2 improvement over the initial convolution when $R \approx \frac{CN}{C+N}$).

The bi-clustering in [Denton et al., 2014] makes a "theoretical" comparison with the complexity of proposed approach problematic, as on the one hand bi-clustering increases the number of tensors to be approximated, but on the other hand, reduces the required ranks considerably (so that assuming the same R would not be reasonable). We therefore restrict ourselves to the empirical comparison.

3.2 Experiments

In this section, the approach is tested on two network architectures, small character-classification CNN and a bigger net trained for ILSVRC. Most of the experiments in this chapter are devoted to the approximation of single layers, when other layers remain intact apart from the fine-tuning.

Several measurements are made to evaluate the models. After the approximation of the convolutional weights with the CP-decomposition, the accuracy of this decomposition is calculated, i.e. $\|W' - W\|/\|W\|$, where W is the original weights and W' is the obtained approximation. The difference between the original weights and approximation may disturb data propagation in CNN, resulting in the drop of classification accuracy. This drop is measured before and after the fine-tuning of CNN. Furthermore, the CPU timings are recorded for the models and the speed-up compared to the CPU timings of the original model are reported (all timings are based on Caffe code run in the CPU mode on image batches of size 64). Finally, the reduction in the number of parameters resulting from the low-rank approximation is calculated. All results are reported for a number of ranks R.

3.2.1 Character-classification CNN

For the experiments with a smaller network, we chose character classification CNN described in [Jaderberg et al., 2014b] as a suitable baseline. The network has four convolutional layers with maxout nonlinearities between them and a softmax output. It was trained to classify 24×24 image patches into one of 36 classes (10 digits plus 26 characters). Our Caffe port of the publicly available pre-trained model (referred below as CharNet) achieves 91.2% accuracy on the test set (very similar to the original). As in [Jaderberg et al., 2014a], the second and third convolutional layer are considered. These layers constitute more than 90% of processing time. Layer 2 has 48 input, and 128 output channels and filters of size 9×9, layer 3 has 64 input and 512 output channels, filter size is 8×8. The results of separate approximation of layers 2 and 3 are shown in Figure 3.2a and Figure 3.2b. Tensor approximation error diminishes with the growth of approximation rank, and when the approximation rank becomes big enough, it is possible to approximate weights accurately. However, our experiments show that accurate approximation is not required for the network to function properly, especially after finetuning. For example, while approximating layer 3, network classification accuracy is unaffected even if the approximation error is as big as 78%.

Since the decomposition greatly reduces number of parameters, it can be also seen as a powerful regularization technique, particularly useful for overparametrized models such as CharNet. Network with decomposed layer overfits less and can reach better accuracy on the test set after some finetuning.

Combining approximations. Decomposition can be applied to multiple layers of a neural network, which is verified by the following experiment. Firstly, layer 2 was approximated with rank 64. After that, the drop in accuracy was made small by fine-tuning of all layers but the new ones. Finally, layer 3 was approximated with rank 64, and for this layer, such approximation does not result in a significant drop of network prediction accuracy, so there is no need to fine-tune the network one more time. The network derived by this procedure is 8.5 times faster than the original model, while classification accuracy drops by 1% to 90.2%. Comparing with [Jaderberg et al., 2014a], CP-decomposition achieves almost two times bigger speedup for the same loss of accuracy ([Jaderberg et al., 2014a] incurs 1% accuracy loss for the speedup of 4.2× and 5% accuracy loss for the speedup of 6×).

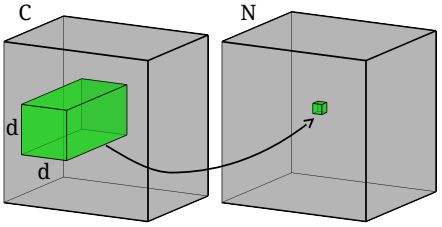
3.2.2 AlexNet

To prove the viability of our method for larger networks and more complicated tasks, we move on to the image classification on ImageNet dataset. Following [Denton et al.,

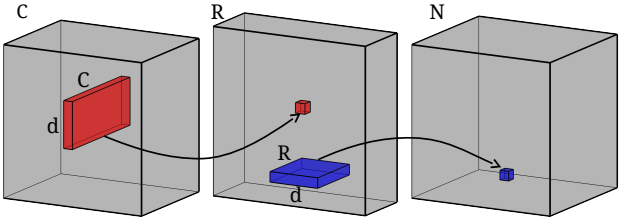
2014], first experiments with AlexNet architecture are performed on the second convolutional layer (filter size 5×5, 96 input and 256 output channels), and pre-trained model shipped with Caffe is used as a baseline. Various network properties for several different ranks of approximation are demonstrated in Figure 3.2c. It can be noticed that conv2 of the considered network demands far larger rank (comparing to the CharNet experiment) for achieving proper performance. Overall, in order to reach the 0.5% accuracy drop reported in [Jaderberg et al., 2014b] it is sufficient to take 200 components, which also gives a superior layer speed-up (3.6× vs. 2×) achieved by Scheme 2 of [Jaderberg et al., 2014b]. The running time of the conv2 can be further reduced with the price of a slight increase in misclassification rate: rank 140 approximation leads to 4.5× speed-up at the cost of $\approx 1\%$ accuracy loss surpassing the results of [Denton et al., 2014]. Along with conventional full-network fine-tuning we tried to refine the obtained tensor approximation by applying the data reconstruction approach from [Jaderberg et al., 2014b]. Unfortunately, we failed to find a good SGD learning rate: larger values led to the exploding gradients, while the smaller ones did not allow to sensibly reduce the reconstruction loss. A probable cause of this effect is the instability of the low-rank CP-decomposition [De Silva and Lim, 2008]. One way to circumvent the issue would be to alternate the components learning (i.e. not optimizing all of them at once), which is the scope of the future work. Another way of dealing with instability follows from the observation that it is not caused by increasing depth of CNN (as proven by the recent advances in training extremely deep networks), but by stacking convolutional layers without nonlinearities between them. Introduction of nonlinearities changes the structure of decomposition, rendering existing methods of calculating low-rank CP-decomposition useless. However, the data-based approach of [Jaderberg et al., 2014b] is still applicable in this setting. Finally, strong regularization also may be able to solve the problem.

3.2.3 NLS vs. Greedy

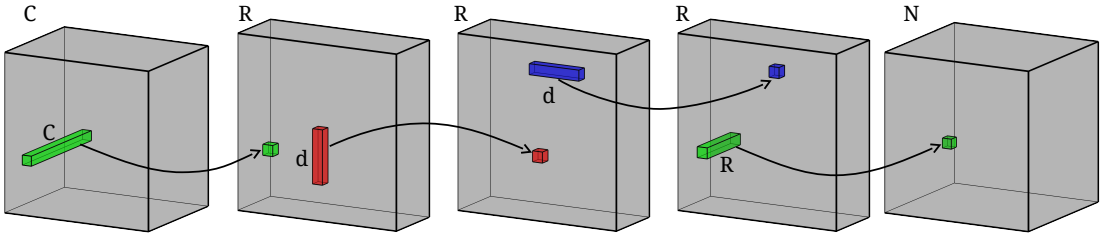
One of the main contributions of this chapter is pointing out that greedy CP-decomposition works worse than more advanced algorithms such as non-linear least squares (NLS), and evaluation of this degradation in the context of speeding up CNNs. Comparisons on the second layers is performed for both CharNet and AlexNet, and for CharNet, the combination with fine-tuning is also evaluated. Additionally, the random initialization using the scheme of [Glorot and Bengio, 2010] was tested for CharNet. The results in Table 3.1 clearly demonstrate two related things. Firstly, NLS decomposition leads to significantly higher accuracy whether with fine-tuning or without. The advantage is greater for the more complex network (AlexNet). Secondly, the output of the fine-tuning clearly depends on the quality of the approximation. This observation concurs with the



(A) Full convolution

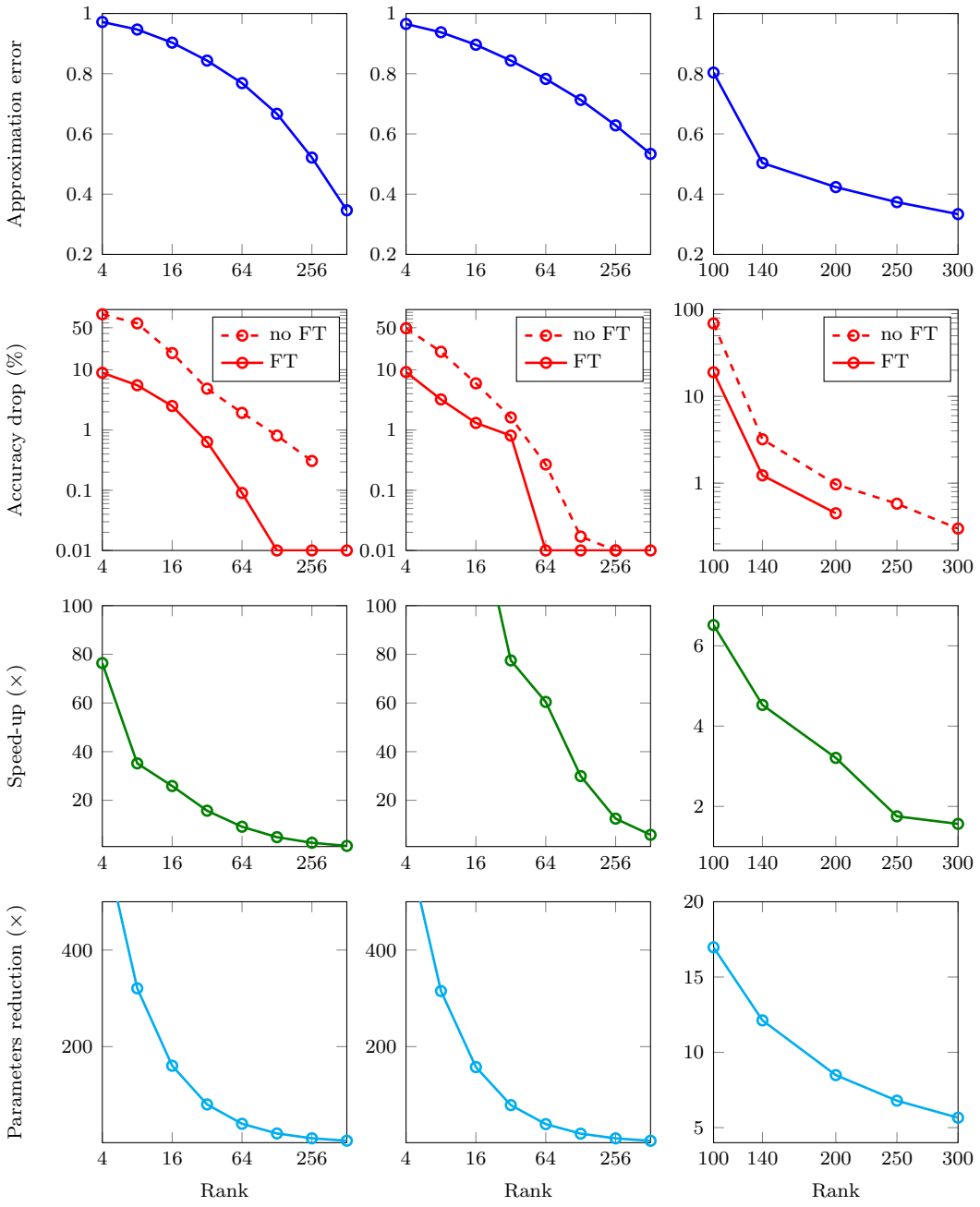


(B) Two component decomposition [Jaderberg et al., 2014b]



(C) CP-decomposition

FIGURE 3.1: Tensor decompositions for speeding up a generalized convolution. Gray boxes correspond to 3D tensors (map stacks) within a CNN, with the two frontal sides corresponding to spatial dimensions. Arrows show linear mappings and demonstrate how scalar values on the right (small boxes corresponding to single elements of the target tensor) are computed. Initial full convolution computes each element of the output array as a linear combination of the elements of a 3D slice that spans a spatial $d \times d$ window over all input maps. [Jaderberg et al., 2014b] approximate the initial convolution as a composition of two linear mappings with the intermediate map stack having R maps (where R is the rank of the decomposition). Each of the two mappings computes each target value based on a spatial window of size $1 \times d$ or $d \times 1$ in all input maps. Finally, CP-decomposition approximates the convolution as a composition of four convolutions with small filters, so that a target value is computed based on a 1D-array that spans either one pixel in all input maps, or a 1D spatial window in one input map



(A) `CharNet conv2` (B) `CharNet conv3` (C) `AlexNet conv2`

FIGURE 3.2: **Various properties and performance metrics of different approximated CNNs plotted as functions of the approximation rank. First row:** weight array approximation error. **Second row:** drop of the classification accuracy of the full model with approximated layers w.r.t the accuracy of the original model; *dashed* lines correspond to the non-tuned CNNs, *solid* lines depict the performance after the fine-tuning. *Note the log-scale. Cases where the accuracy has actually improved are plotted at the bottom line.* **Third row:** empirical speed-up of the approximated layer w.r.t. the original layer. **Fourth row:** ratio between the numbers of parameters in the original and the approximated layers.

	CharNet,R=16		CharNet,R=64		CharNet,R=256		AlexNet	AlexNet	AlexNet
	No FT	FT	No FT	FT	No FT	FT	R=140	R=200	R=300
Random	–	9.70	–	7.64	–	6.13	–	–	–
Greedy	24.15	2.64	4.99	0.46	1.14	-0.31	65.04	7.29	4.76
NLS	18.96	2.16	1.93	0.09	0.31	-0.52	3.21	0.97	0.30

TABLE 3.1: Accuracy drop for the greedy and the non-linear least-squares (NLS) CP-decomposition. The results are given for the second layers of CharNet and AlexNet, for different decomposition ranks R, and the numbers correspond to the accuracy drop of the entire CNN. Original networks achieve 91.24% (CharNet) and 79.95% (AlexNet). For AlexNet, results are presented without finetuning, and for CharNet the effect of fine-tuning (FT) is also evaluated. NLS computation of the CP-decomposition invariably leads to better performance, and the advantage persists through fine-tuning.

hypothesis that the large number of parameters in CNN is needed to avoid poor local minima. Indeed, CP-decomposition radically decreases the number of parameters in a layer. While good minima may still exist (e.g. the NLS+FT result), optimization is prone to sticking in a much worse minima (e.g. the Random+FT result).

3.3 Conclusion

This chapter demonstrated that a rather straightforward application of a tensor decomposition method (NLS-based low-rank CP-decomposition) combined with the discriminative fine-tuning of the entire network can achieve considerable speedups with minimal loss in accuracy. Additionally, good classification performance on the low decomposition ranks corresponding to large approximation error demonstrated low-rank structure presented in the convolutional weights.

However, the finetuning of the decomposed model is complicated by the instability of the decomposition. In the experiments of this chapter, we have successfully avoided this problem by fixing the subset of layers during the fine-tuning stage, which may limit the final performance. This limit becomes tighter as more and more layers are decomposed.

The proposed method is particularly effective in case of relatively shallow neural networks with large filters, as demonstrated in the experimental section with the CharNet example. Unfortunately, modern CNNs evolved in the direction of smaller filters and larger depth, severely narrowing the practical applications of CP-decomposition method.

The perfect approach for speeding up the neural network would allow to simultaneously speed up as many layers as needed and finetune the whole network with the ease of the original training process. In the next chapter, we present a different approach which has the desired properties.

Chapter 4

Group-wise Brain Damage

The original *Optimal Brain Damage* work [LeCun et al., 1990] describes a carefully designed "brain-damage" process that can prune the coefficients of a multi-layer neural network very significantly while incurring minimal or no loss of the prediction accuracy. Such process resembles the biological learning processes in mammals, in whose brains the number of synapses peak during early childhood and is then reduced substantially in the process of *synaptic pruning* [Chechik et al., 1998]. The optimal brain damage algorithm and its variants, however, impose sparsity in an unstructured way. As a result, while a large number of parameters can be pruned, the attained level of sparsity in the network is usually insufficient to achieve substantial computational speedup on modern architectures.

This chapter presents a simple approach that modifies the standard generalized convolution process by imposing *structured* "brain-damage" on the convolutional weights. This approach, combined with a particular choice of structure, allows obtaining considerable speed-up of convolutional layers in neural networks.

This structure is motivated by the observation that the majority of current implementations of generalized convolutions (including the most efficient one at the time when experiments presented in this chapter were performed) [Chellapilla et al., 2006, Donahue et al., 2014, Jia et al., 2014a, Chetlur et al., 2014, Vedaldi and Lenc, 2014, NervanaSystems, 2015] compute generalized convolutions by reducing them to matrix multiplications (this reduction is also referred to as *unrolling*, or the `im2col` operation, and was described in detail Section 1.3). While unstructured brain damage in a convolutional layer, i.e. shrinking some of the convolutional weights to zero, will make one of the factor matrices (*the filter matrix*) sparse, it will not make the overall multiplication run faster. The idea therefore is to group together the entries of convolutional weights array in a certain fashion and to shrink such groups to zero in a coordinated way. By doing

this, one can eliminate rows and columns from both factor matrices that are multiplied when convolution is reduced to matrix multiplication. Repeated elimination of rows and columns makes both factor matrices thinner (but still dense) and results in faster matrix multiplication.

Conventional group sparsity regularizer [Yuan and Lin, 2006] embedded into stochastic gradient descent minimization is able to accomplish group-wise brain damage efficiently. The use of group sparsity thus allows optimizing receptive fields in the convolutional network. This approach makes the case for the natural idea of using structured sparsity as a simple way to optimize connectivity in deep architectures. In our experiments, speed-up factors exceed those obtained by tensor-factorization based methods. For example, experiments show that group-wise brain damage can accelerate the bottleneck layers of AlexNet ('conv2' and 'conv3') by a factor of 8.5 $\times$ simultaneously while incurring only modest (1%) loss of the prediction accuracy.

4.1 Method

4.1.1 Group-Sparse Convolutions

In the section below, the reduction from generalized convolution to matrix multiplication introduced in Section 1.3 is reviewed in more details, and itss implications to the sparsity structure are discussed.

Generalized convolution within a convolutional layer transforms an input stack of C maps of size $W' \times H'$, which can be treated as a three-dimensional array U_{whc}, into an output stack of N maps of size $W'' \times H''$ which form a three-dimensional array V_{whn}. The exact relation between W', H' and W'', H'' depends on the padding and stride settings within the layer, and the approach presented in this chapter can handle any padding/striding settings seamlessly. The transformation is defined by the following formula:

$$V(x,y,k) = \sum_{c=1}^{C} \sum_{\substack{i=1..d \\ j=1..d}} W(i,j,c,k) U(x+i, y+j, c) \tag{4.1}$$

Here, W is a four-dimensional array of convolutional weights (kernel tensor) of size $d \times d \times C \times N$ with the first two dimensions corresponding to the spatial dimensions, the third dimension corresponding to input maps, the fourth dimension corresponding to

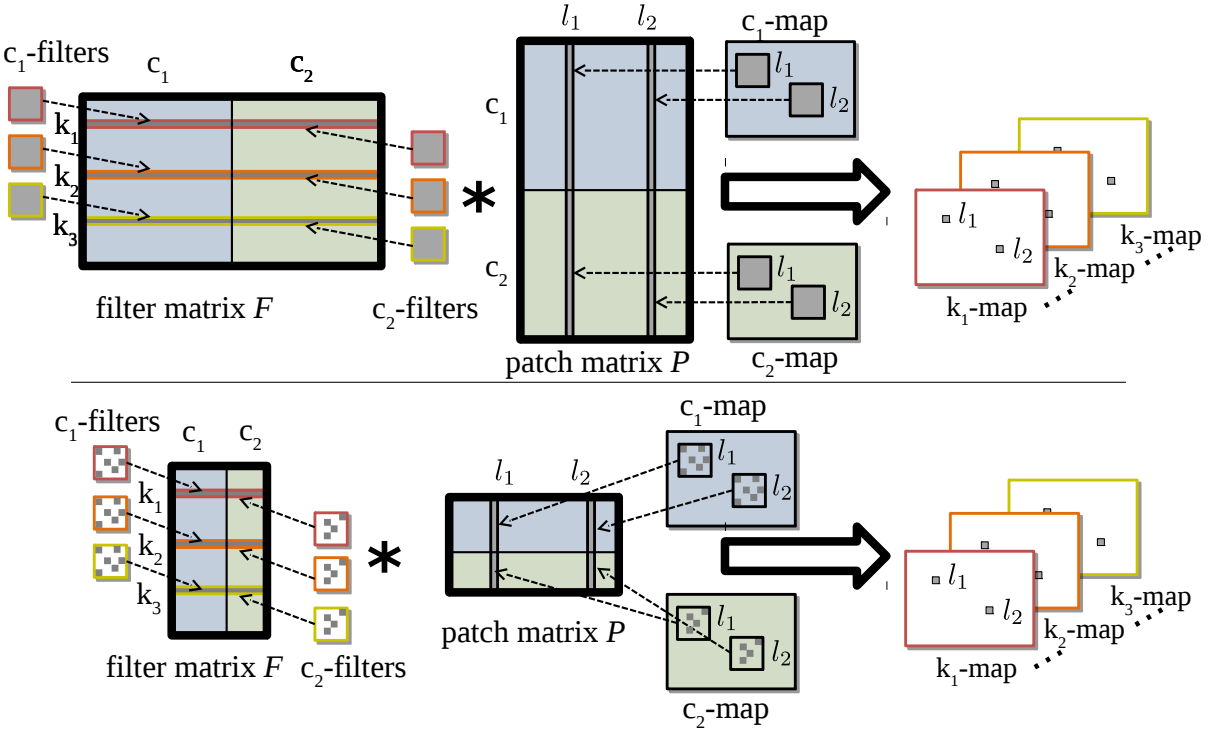


FIGURE 4.1: Standard Generalized Convolution (top) vs. Generalized Convolution after Group-wise Brain Damage (bottom). In both cases, diagram for two input maps are shown ($C = 2$, blue-green color coding), and three output maps are highlighted (k_1, k_2, k_3 color-coded red-orange-yellow). Also, two stacial locations l_1 and l_2 are highlighted. In both cases, the output map stack is obtained by reshaping the product of the filter matrix and the patch matrix. In the standard case, the filters and the patches sampled during the formation of the patch matrix are dense. After group-wise brain damage, both the filters and the patch sampling patterns are group-sparse (one sparsity pattern per input map), which results in a much thinner filter and patch matrices and thus leads to much faster matrix multiplication/convolution.

output maps. The spatial width and height of the kernel are denoted as d (for simplicity, square-shaped kernels and odd d are assumed).

The implementation of (4.1) can be reduced to the multiplication of two dense matrixes [Chellapilla et al., 2006]. The reduction proceeds as follows:

- The weights array W is transformed into the *filter matrix* F of size $N \times d^2C$ by mapping a sequence of C flattened 2D filters $W(:, :, c, k)$ into a k-th row of the matrix.
- The input map stack U is reshaped into the *patch matrix* P of size $d^2C \times W''H''$, where the l-th column corresponds to a certain output location $l = (x, y)$ and is stacked from the C patches extracted from C input maps, all centered at this location and reshaped in a row-wise fashion into column vectors.

- The filter matrix F is multiplied by the patch matrix P resulting in a matrix $\tilde{V}$ of size $N \times W''H''$ that contains all the elements of V (each column corresponds to a certain location and contains the values of this location in the N output maps). The multiplication implements (4.1) exactly, as each row-by-column product within the multiplication corresponds to one instance of the computation (4.1) for certain (x, y, k). The output array (map stack) V can be obtained from $\tilde{V}$ by reshaping.

The construction discussed above has proven to be highly successful and is used in the majority of modern CNNs "backends", e.g. [Chellapilla et al., 2006, Donahue et al., 2014, Jia et al., 2014a, Chetlur et al., 2014, Vedaldi and Lenc, 2014, NervanaSystems, 2015]. The key idea is to train CNNs with sparse convolutional kernels that are consistent with this construction.

Such consistency can be achieved if the sparsity patterns are aligned in a certain way. Formally, group-wise brain damage introduces a *sparsity pattern* Q_c for every input map $c \in 1 \dots C$. The sparsity pattern is defined as a subset of the full spatial d-by-d grid, i.e. $Q_c \subset \{1 \dots d\} \otimes \{1 \dots d\}$. The convolutional operation then becomes a slight modification of (4.1):

$$V(x, y, k) = \sum_{c=1}^{C} \sum_{(i,j) \in Q_c} W(i, j, c, k) U(x+i, y+j, c) \tag{4.2}$$

The reduction of (4.2) is an almost straightforward replication of the procedure [Chellapilla et al., 2006]. The only modifications are (Figure 4.1):

- When the filter matrix is assembled, each 2D filter $W(:, :, c, k)$ is reshaped into a row-vector of length $|Q_c|$ by including only non-zero elements. The filter matrix thus becomes of size $N \times \sum_{c=1}^{C} |Q_c|$.
- When the patch matrix is assembled, each 2D patch at location $l = (x, y)$ in map c is reshaped into a column vector of size $|Q_c|$ by sampling the input map $U(:, :, c)$ sparsely at locations $(x+i-\frac{d+1}{2}, y+j-\frac{d+1}{2})$, where $(i, j) \in Q_c$. The patch matrix thus becomes of size $\sum_{c=1}^{C} |Q_c| \times W''H''$.

As a result of this modification, the multiplication of two dense matrices of sizes $T \times d^2C$ and $d^2C \times W''H''$ is replaced by the multiplication of two dense matrices of sizes $N \times \sum_{c=1}^{C} |Q_c|$ and $\sum_{c=1}^{S} |Q_c| \times W''H''$, which results in the $d^2C / \sum_{c=1}^{C} |Q_c|$-times reduction in the number of scalar operations. In our experiments with the reference implementation of [Jia et al., 2014b] the wall-clock reduction in the convolution time between

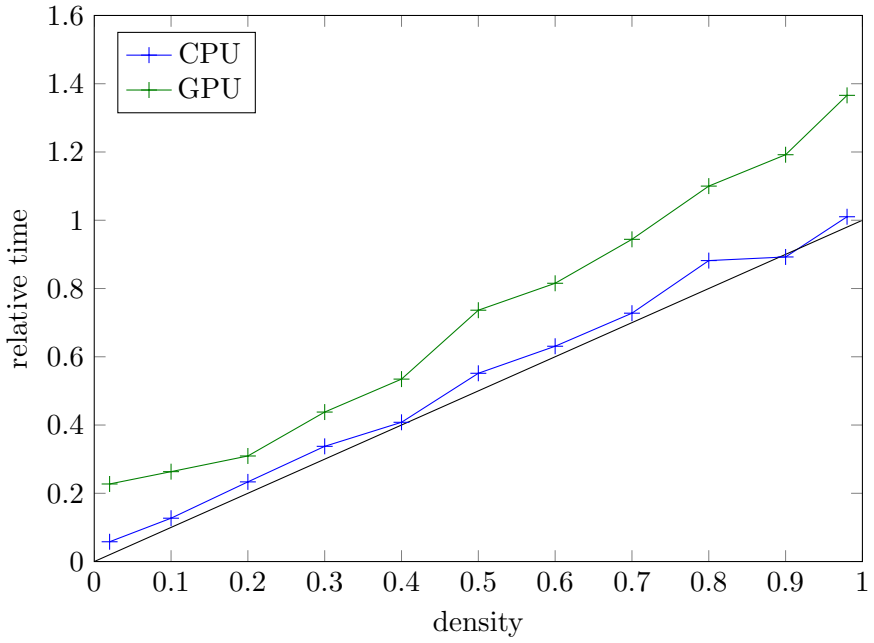


FIGURE 4.2: The blue curve shows speed-up versus density level τ, measured for forward propagation in the second convolutional layer of AlexNet on CPU (Intel Core i7-4960X) and GPU (GeForce GTX TITAN Black). The measurements are averaged over 100 runs, and speed-up is reported relative to the original dense layer, so the value larger than 1.0 corresponds to slower evaluation. Importantly, the observed speedup is almost linear in the sparsity level (diagonal). The green curve shows the layer speedup on a GPU. While a certain constant overhead can be seen, we believe that (part of) it can be eliminated via more elaborated GPU implementation.

the original implementation and the group-sparse convolution was almost matching the "theoretical" speed-up factor (Figure 4.2).

The idea of group-sparse convolution can be applied to obtain fast CNNs in at least two ways. First, the fast version of the network can be trained from scratch, and secondly, it can be obtained by modification of pretrained architectures (i.e. performing "brain damage").

4.1.2 Fixed sparsity pattern

Predefined group-sparsity pattern. The simplest solution considered here is to choose the sparsity patterns Ω_S in advance in a data-independent manner, and enforce these patterns during the learning of the network. One particular case of this approach is a simple reduction of the spatial size of filters to a minimum, e.g. three-by-three, or even smaller rectangular pattern all the way to one-by-one (this is in line with a recent work of [He and Sun, 2015] where they consider 2×2 filters for some of their architectures). Note, that structured sparsity approach allows for all kinds of non-rectangular filters, and it becomes very useful in the experiments.

One of the downsides of this approach is that when designing an architecture with multiple convolution layers, there are no clear design principles that can guide the choice of the filter shapes. In contrast, the methods discussed below can start with larger filters and then shrink their sizes towards optimally-shaped small filters. Furthermore, the restriction of the filter size can make learning process susceptible to poorer local minima. Even when good architectures with very sparse filters exist, it may not be possible to reach them by gradient descent without first considering a less-constrained more parameter-reach architecture with large dense filters.

Training with group-sparsity regularizer. Rather than fixing the group-sparsity pattern in advance, it is possible to find it as a part of learning process while the network is trained. A classical way to achieve this is through the use of group-sparsity regularization [Yuan and Lin, 2006, Roth and Fischer, 2008, Jenatton et al., 2011]. Let us consider a regularizer based on $l_{2,1}$-norm:

$$\Omega_{2,1}(W) = \lambda \sum_{i,j,c} \|\Gamma_{ijc}\| = \lambda \sum_{i,j,c} \sqrt{\sum_{k=1}^{N} W(i,j,c,k)^2}\,, \tag{4.3}$$

where the vector Γ_{ijc} denotes the group of the weights array entries $W(i,j,c,:)$. The effect of the regularizer (4.3) is in shrinking some of such groups to zero in a coordinated fashion. When an entire group Γ_{ijc} is set to zero, one can set the pixel (i,j) in the sparsity pattern Ω_c to zero, thus increasing the group-sparsity.

For a convolutional layer that is being pruned, the gradient of (4.3), i.e.:

$$\frac{\partial \Omega_{2,1}(W)}{\partial W(i,j,c,k)} = \lambda \frac{W(i,j,c,k)}{\sqrt{\sum_{z=1}^{N} W(i,j,c,z)^2}} \tag{4.4}$$

can simply be added to the gradient of the learning loss while performing stochastic gradient updates in the course of learning. The coefficient λ in (4.3) and (4.4) controls the strength of the regularization w.r.t. the main learning loss.

Generally, using the regularizer (4.3) will result in a group-sparsified weights array with some of Γ_{ijc} having only near-zero entries. Because of the stochastic nature of SGD and non-differentiability of l_{12} norm near zero, the entries in these groups will not be exactly zero, and further postprocessing is needed to nullify the near-zero groups and to set the sparsity patterns $\Omega_C f$ accordingly.

4.1.3 Sparsifying with Group-wise Brain Damage

While it is possible to train CNNs with group-sparse convolutions from scratch, the main focus of this work is developing algorithms that can speed-up existing pretrained networks, as these networks take excessive time for training. Towards this end, two approaches have been developed. The approaches that can accelerate pretrained networks by inflicting group-wise brain damage in a way that the drop in the prediction accuracy is kept small.

Group-wise sparsification with fine-tuning. The implementation of this approach is also based on the group-sparse regularizer (4.3). It starts with the input CNN and runs the learning process on the dataset D with the added regularizer (4.3). After a certain amount of iterations, a predefined number of groups Γ_{ijc} with the smallest $l2$-norm is set to zero. For a desired density level $\tau \in [0, 1]$ and respective speedup $1/\tau$, $d^2C\,(1-\tau)$ groups are set to zero, making the respective Q_C sparse.

This approach has two complications. Firstly, for a given density τ it was generally hard to set appropriate regularization strength λ in advance without trying several values. Secondly, for small τ (large speedup) the appropriate regularization strength λ typically leads to an excessive regularization, as many groups end up being biased towards zero but not close to zero. Because of that, the prediction accuracy for such λ experienced significant drop in the process of learning as compared to the input CNN.

Fortunately, one can recover from most of this drop by the subsequent *fine-tuning* of the network, that follows after the brain-damage process. For the fine-tuning, the sparsity patterns Q_C was fixed and the learning process was resumed without group-sparse regularization, for an excessive number of epochs. As a result of such fine-tuning, the network adapts to the imposed sparsity patterns, while the prediction accuracy goes up and recovers most of the drop.

Gradual group-wise sparsification. Two complications discussed above can be avoided with an alternative approach, which combines the brain-damage and the fine-tuning processes, and furthermore avoids most of the need for a manual search for good meta-parameter values. The approach also often leads to considerably better results.

A new kind of regularizer was developed for this approach, called *truncated* l_{12} regularizer:

$$\Omega^T_{2,1}(K) = \lambda \sum_{i,j,c} \min(\|\Gamma_{ijc}\|, \theta) \tag{4.5}$$

The gradient of (4.5) equals (4.4) when $\|\Gamma_{ijc}\| < \theta$ and is zero otherwise. Informally speaking, the value of θ controls which groups are considered "promising" and are being

shrinked towards zero, and which groups are considered to be too far from zero and therefore stay unaffected by the regularizer (4.5).

A special validation set is created to monitor the performance of the network during brain damage process, and maximum tolerable accuracy drop δ on this dataset is chosen. The training process starts with the pretrained network, and the regularizer (4.5) is applied with varying θ. Specifically, θ is increased (intensifying brain damage) if the accuracy drop on hold-out set is less than δ and decreased, thus relieving certain groups from the effect of the regularizer, if the drop is greater than δ.

To perform the actual sparsification, an additional threshold $\epsilon \ll \delta$ is introduced. In the process of learning, when the norm of a certain group falls below the threshold (i.e. $\|\Gamma_{ijc}\| < \epsilon$) the group is greedily fixed to zero and eliminated from the array. The sparsity thus monotonically increases through the process, and training is carried on until the sparsification process stalls, i.e. the system keeps training with Γ and performance drop oscillating, while no new groups have their lengths fall under ϵ for a number of epochs. In all performed experiments, all increments and decrements of θ were based on five-percent quantiles of the groups. I.e. every time θ is adjusted in a steps to bring 5% of groups Γ_{ijc} in or out of the $\|\Gamma_{ijc}\| < \theta$ "territory".

Overall, we find the whole procedure to be rather insensitive to the choices of λ and ϵ. It is more practical, and it often leads to higher sparsity levels and speed-ups than those attainable by the pruning with the fine-tuning approach. Most importantly, one could use the same λ and ϵ, as well as the same shared value of θ when pruning several layers simultaneously.

Hovewer, in case of deep architectures, simultaneous pruning is complicated by the fact that some layers are inevitably pruned faster than the others. When the average achieved density is sufficiently small, these layers are left allmost empty, creating a bottlenecks inside the network and effectively stopping the pruning in the rest of the network. We deal with this problem by implementing rescaling described in [Neyshabur et al., 2015].

Since the CNNs are build of linear operations, the ouputs of the network are not going to change if weights of l-th convolutional layer ar multiplied by the constant k_l, given that

$$\prod_{i=l}^{n} k_l = 1 \tag{4.6}$$

where n is the number of the layers participating in the rescaling. We set $k_l = \frac{1}{max\|\Gamma_{ijc}\|}$, then normalize the set of k_l to satisfy (4.6) and multiply the convolutional weights by

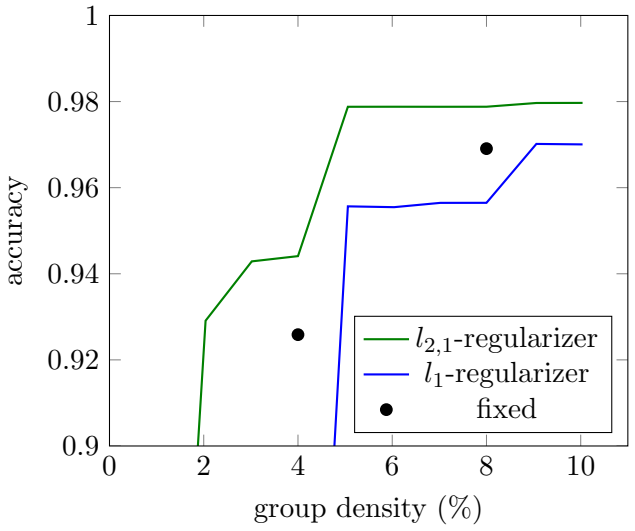


FIGURE 4.3: Accuracy vs. density level on MNIST dataset (LeNet architecture) for various CNNs with group-sparse convolutions. The following approaches are compared: training with $l_{2,1}$ and l_1 regularizations followed by sparsifications, and training with predefined sparsity patterns Ω_S (black dots). Overall, training with $l_{2,1}$ regularizer obtains the best result that can be further improved by fine-tuning without regularization.

the corresponding k_l. Repeated every several training iterations, this procedure allows to avoid bottleneck generation.

4.2 Experiments

Implementation details. Our implementation is based on `Caffe` [Jia et al., 2014b] and modifies their original convolution, which is implemented as two subsequent transformations (`im2col` and matrix multiplication). To implement the group-sparse convolution we focused on the forward propagation step and CPU computation. Most of the methods can be extended for backprop step and for GPUs, however making such extensions efficient is non-trivial. Experiments presented in this chapter only require the modification of `im2col` function, so that it can fill in the patch matrix while following certain sparsity patterns.

Datasets. The following experiments are performed. Firstly, training CNNs with group-wise brain damage and from scratch is compared with baselines in a small-scale setting. We use MNIST dataset [Lecun et al., 1998] for these small-scale experiments. Next, a large-scale problem is considered, namely ImageNet (ILSVRC) image classification and the task of accelerating of a pretrained architecture, namely the Caffe version of AlexNet [Krizhevsky et al., 2012]. Additionally, some results for one of the VGGNet networks [Simonyan and Zisserman, 2015] are presented.

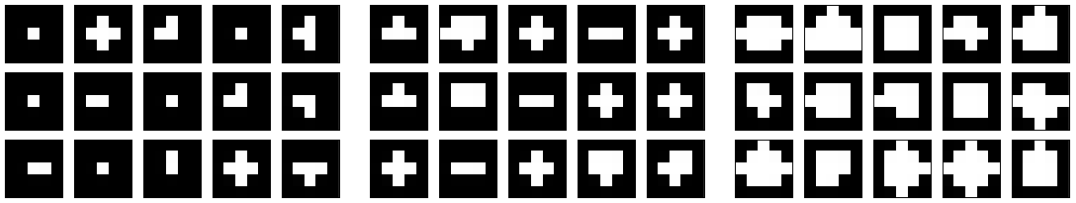
method	density	speed-up	accuracy drop
Accelerating the second convolutional layer of AlexNet			
Denton et al. [2014] Tensor decomposition + Fine-tuning		2.7x	$\sim 1\%$
Lebedev et al. [2015] CP-decomposition + Fine-tuning		4.5x	$\sim 1\%$
Jaderberg et al. [2014b] Tensor decomposition + Fine-tuning		6.6x	$\sim 1\%$
Training with fixed sparsity patterns	0.12	8.33x	0.82%
Training with fixed sparsity patterns	0.2	5x	0.16%
Group-wise sparsification + Fine-tuning	0.1	10x	1.13%
Group-wise sparsification + Fine-tuning	0.2	5x	0.43%
Group-wise sparsification + Fine-tuning	0.3	3.33x	0.11%
Group-wise sparsification + Fine-tuning	0.4	2.5x	-0.09%
Gradual group-wise sparsification	0.11	9.0x	0.28%
Gradual group-wise sparsification	0.05	20x	1.07%
Accelerating the second and the third convolutional layers of AlexNet			
Training with fixed sparsity patterns	0.12	8.7x	1.54%
Training with fixed sparsity patterns	0.35	2.9x	0.36%
Training with fixed sparsity patterns	0.54	1.9x	-0.53%
Group-wise sparsification + Fine-tuning	0.2	5x	1.50%
Group-wise sparsification + Fine-tuning	0.3	3.33x	1.17%
Group-wise sparsification + Fine-tuning	0.5	2x	0.57%
Gradual group-wise sparsification	0.12	8.5x	1.04%
Accelerating all five convolutional layers of AlexNet			
Training with fixed sparsity patterns	0.34	3.0x	1.34%
Gradual group-wise sparsification	0.31	3.2x	1.43%

TABLE 4.1: **Accelerating convolutional layers of the pretrained AlexNet architecture**: results of the two variants of the method for various sparsity levels alongside tensor-decomposition based methods (note: the results for [Jaderberg et al., 2014b] are reproduced from [Lebedev et al., 2015]).

4.2.1 MNIST experiments

LeNet architecture was chosen for small-scale MNIST experiments. We trained the LeNet architecture on the MNIST dataset from random initialization while adding the group-sparse regularization (section 4.1.2) while varying the regularization strength λ and picking the optimal one for each sparsity level. The sparsification affects both convolutional layers of LeNet, and the same density level τ is enforced in both layers. A number of baselines were also considered:

- A simple baseline that trains the network without regularization and then simply eliminates (sets to zero) a certain number of groups Γ_{ijc} with the smallest l2-norms. The performance of this baseline was clearly below all other methods and it is not reported.



(A) sparsity $1 - \tau = 0.9$ (B) sparsity $1 - \tau = 0.8$ (C) sparsity $1 - \tau = 0.6$

FIGURE 4.4: The sparsity patterns obtained by group-wise brain damage on the second convolutional layer of AlexNet for different sparsity levels. Nonzero weights are shown in white. In general, group-wise brain damage shrinks the receptive fields towards the center and tends to make them circular.

- Picking sparsity patterns Q_S in advance. We consider filters with only one central non-zero entry and filters with two adjacent central non-zero elements. These options correspond to the density of 4% and 8% respectively. The former is essentially equivalent to a non-convolutional network, where almost all processing happens in the fully-connected layers.

- Another baseline is a simpler non-group-wise sparsification obtained by training with l1-norm regularizer (with varying λ) but then nullifying groups $|\Gamma_{ijc}$ based on their norms.

The results of the proposed method and the baselines are shown in Figure 4.3. The rightmost plot shows the comparison of the l_1-envelope, $l_{2,1}$-envelope, and the performance of the group-wise brain damage applied to the network trained without sparsity-inducing regularizer. The use of group-sparsity regularization boosts the performance of group-wise brain damage very considerably. Twenty-fold acceleration of convolutional layers can be obtained while keeping the error low (2.1%, reduced to 1.71% after fine-tuning). Using l_1-regularizer followed by optimal brain damage works worse than $l_{2,1}$-regularizer. Pre-fixing sparsity patterns achieves good results, which are still worse than training with group-sparsity regularizer. Note also that all methods except the baseline with the pre-fixed patterns can be improved via fine-tuning.

4.2.2 ILSVRC experiments

AlexNet (Caffe reimplementation) architecture that has five convolutional layers. In this section, the following subtasks are considered: (i) accelerating the second convolutional layer (which is the slowest of all layers), (ii) accelerating the second and the third layers (which are the two slowest layers), (iii) accelerating all five convolutional layers (which together take the vast majority of the forward-propagation time). When reporting the

final density in subtasks (ii) and (iii), we weigh the densities in different layers by the forward propagation times.

As an additional baseline, we evaluate the variant of the method that trims the network according to some predefined sparsity pattern and then learns the network while keeping the same fixed pattern. Namely, the following symmetric centered patterns are considered: vertical or horizontal block 1×3, the 3×3 cross pattern, 3×3 square or diamond shape inside 5×5 filter.

For the first two subtasks, we evaluate the method sparsity-inducing regularizer for various sparsity levels. For every desired density level τ, the optimal value of λ have to searched in large range with ten-fold increments. For each τ we pick λ that results in the minimal accuracy drop after pruning before fine-tuning. After picking the optimal λ, fine-tuning is performed with fixed sparsity patterns. Figure 4.4 demonstrate sparsity patterns Ω_S obtained for different sparsity levels.

Finally, for all three subtasks, we evaluate the most advanced of presented methods, namely gradual group-wise pruning. The parameters λ and ϵ are set to 0.01 and 0.1 respectively, and the evaluation set of ILSVRC2012 is split randomly into two halves. One half is used solely to estimate the drop of the classification accuracy in the dynamical adjustment of θ, and the other serves for the estimation of the final performance drop. Acceptable performance drop is chosen to be 1% of top-1 accuracy, the same as in the previous chapter.

As shown in Table 4.1, the results of gradual sparsification outperform the tensor factorization methods as well as sparsification with fine-tuning considerably, achieving higher group-sparsification/speed-up for similar prediction accuracy drop. Notably, the proposed approach is more successful in speeding-up AlexNet than a number of approaches based on tensor decomposition. Figure 4.5 further visualizes the process of the simultaneous gradual brain damage inflicted on all five layers of AlexNet.

"External" computer vision task. Convolutional layers of large networks pretrained on large annotated training sets such as ILSVRC can be used as universal spatially localized features in a variety of ways [Liu et al., 2015b, Azizpour et al., 2015], which is particularly valuable for problems with considerably smaller training sets. Recently, [Babenko and Lempitsky, 2015] showed that descriptors obtained by sum-pooling of the features that emerge in the last convolutional layer of a pretrained network can be used as state-of-the-art holistic descriptors for image retrieval. Comparison of AlexNet as a base model and the network with the simultaneous group-sparsification of all convolutional layers from Table 4.1 with $3.2x$ speedup revealed a negligible drop in performance for the INRIA holidays dataset [Jégou et al., 2008] from 0.783 mAP to 0.780 mAP, and a

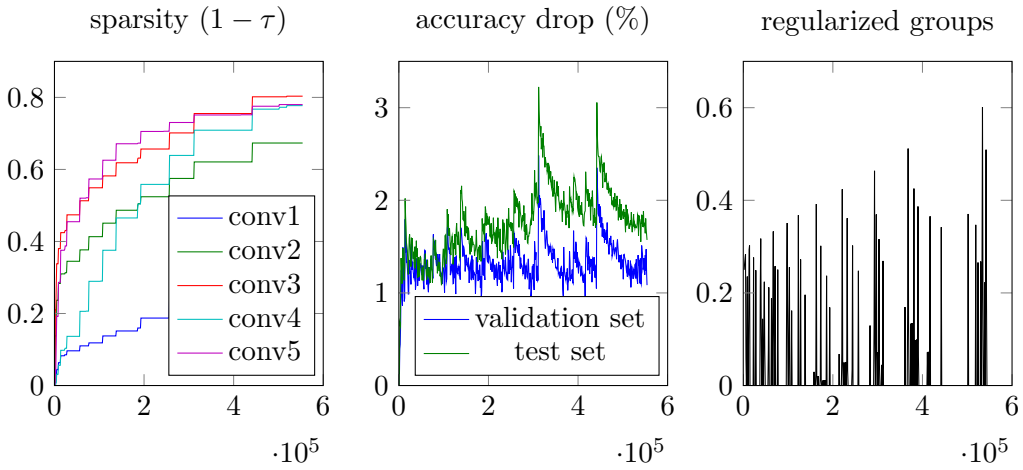


FIGURE 4.5: The process of sparsification of all five layers in AlexNet. The left plot shows the monotonic growth of the sparsity levels of the five convolutional layers as the iterations progress. The middle plot shows the relative prediction accuracy drop for the current system for the validation part and for the hold-out test set. Finally, the right part visualizes the process of the adjustment of θ threshold in the truncated $l_{2,1}$ regularization. This plot shows the percentile of groups Γ_{ijc} with the l_2-norm less than θ. θ is increased or decreased dependent on whether the performance drop on the validation set is greater or smaller than 1.2%.

reasonably small drop for the Oxford Building dataset [Philbin et al., 2007] from 0.45 to 0.41.

VGGNet results. We have also applied the gradual group-wise sparsification to the slowest convolutional layer of VGGNet (the deeper 19 layer version of [Simonyan and Zisserman, 2015], starting from its Caffe Zoo version. The sparsification obtained the density $\tau = 0.13$ with only 0.2% top-1 accuracy drop. Interestingly, unlike the experiments with AlexNet where empty sparsity patterns Ω_S ("dead feature maps") are rare, in this example such all-zero patterns were present (29 out of 64), suggesting that this manually designed architecture contains an excessive number of feature maps in this layer. This result also suggests that pruning is suitable even for networks with very small initial filter sizes in convolutional layers (3×3 for VGGNet). When applied to all convolutional layers of the VGG network, the method obtains different densities for different layers, creating undesirable bottlenecks in the network that slow down the learning process. With help of rescaling, uniform sparsification reached density $\tau = 0.45$, corresponding to more than $2\times$ speedup with 0.7% accuracy drop.

4.3 Conclusion

In this chapter, we have described a novel approach for speeding up CNNs that uses the group-wise brain damage process to prune the convolutional filters. The method can

be applied to the specific layers or for the whole network simultaneously. The approach takes into account the way generalized convolutions are reduced to matrix multiplications and removes the entries of the convolution kernel in a group-wise fashion. The exact sparsity patterns can be learned from data using group-sparsity regularization. When applied after learning with such regularization and followed by fine-tuning, group-wise brain damage obtains state-of-the-art performance for speeding up CNNs.

Filter shapes, discovered by the pruning process, are roughly circular in most cases. This observation could be taken into account in the future architecture design work. Also, the process treated AlexNet and VGGNet differently, eliminating entire feature maps in the latter case, which demonstrates that the proposed method can in practical setting mimic different types of structural constraints described in Section 2.5, thus eroding the border between pruning and architecture search approaches.

Chapter 5

Impostor Nets

The ability to perform fine-grained recognition is one of the hallmarks of the recent progress in deep learning. The best of fine-grained classifiers [Cai et al., 2017, Zheng et al., 2017, Kong and Fowlkes, 2017, He and Peng, 2017], however, use very deep convolutional networks (CNNs) such as those based on the VGG-architecture [Simonyan and Zisserman, 2015], which means that they are ill-suited for the deployment on mobile platforms and other platforms that lack GPUs, unless each image is processed remotely. At the same time, having a fine-grained classifier "in your pocket" and without the need for a remote server connection is what makes such classifiers particularly useful.

A natural question is then, whether it is necessary to have a very big and deep CNN typically designed for large-scale visual recognition, in order to perform fine-grained classification? To address this question, this chapter focuses on building fine-grained classifiers that perform well and yet do not require a deep, computation- and power-hungry CNN to perform classification well.

Towards this goal, we suggest a new architecture. This architecture combines a compact CNN with a non-parametric classifier, squeezing a maximal performance from such combination. This combination is natural, as the non-parametric classifier is able to compensate for the inability of a compact CNN to achieve linear separation of similar visual classes.

The non-parametric classifier that is used in the proposed system is a radial basis function classifier, which takes the high-dimensional output of an underlying CNN and then performs classification by combining proximity-based votes from a set of points in the embedding space. The described architecture is thus similar to the RBF-solver architecture recently proposed in [Meyer et al., 2017], and also reminiscent of many works on metric learning (as several works evaluate k-NN classifiers on top of the learned metrics).

In the above-mentioned approaches, the voting points are the mappings of the training examples by the learned embedding networks. The distinguishing property of the proposed approach from both [Meyer et al., 2017] and metric learning approaches, is that in this case the voting points are not tied to the training samples. Instead the voting points are initialized to the images of the training examples under the CNN mapping, but drift away from such initialization as the learning progresses. Extra flexibility resulting from the lack of ties between the training examples and the voting points results in a significant boost of the classification accuracy.

The evaluation is performed on two popular fine-grained datasets (Caltech-UCSD Birds [Wah et al., 2011] and Stanford Cars [Krause et al., 2013]). To diversify the evaluation, we also perform experiments on the Landmarks-clean dataset [Gordo et al., 2017] of landmark images, treating landmark recognition as a classification problem [Li et al., 2009]. In all cases, the classification accuracy of an underlying moderately-sized network (SqueezeNet [Iandola et al., 2016] in most of the performed experiments) is boosted considerably using proposed approach, while the additional computation cost is minimal. we also validate the idea that the memory overhead over the baseline CNN can be decreased using standard compression schemes without affecting the classification accuracy strongly.

Finally, the open-set ability of impostor networks is validated. The performed experiments demonstrate that impostor network can identify images that do not belong to training classes, and find that this ability also exceeds the standard classification networks.

5.1 Method

An *impostor network* is an image classifier consisting of a convolutional network f_θ with learnable parameters θ that maps an input image $\mathbf{x}$ into a d-dimensional space $\mathbf{Y}$ as well as a dataset of reference points $\mathbf{c}_1, \ldots, \mathbf{c}_M$ in $\mathbf{Y}$ with assigned class labels $l_1, \ldots, l_M$ that define class kernel densities in $\mathbf{Y}$.

A trained impostor network classifies an image $\mathbf{x}$ by first mapping it to $\mathbf{Y}$:

$$\mathbf{y} = f_\theta(\mathbf{x}) \tag{5.1}$$

and then computing a set of weights $w_1, \ldots, w_M$ using the Gaussian kernel $g(\cdot, \cdot; \sigma)$ traditionally used in the RBF networks:

$$w_j = g(\mathbf{y}, \mathbf{c}; \sigma) = \exp\left(-\frac{\|\mathbf{y} - \mathbf{c}_j\|^2}{2\sigma^2}\right), \tag{5.2}$$

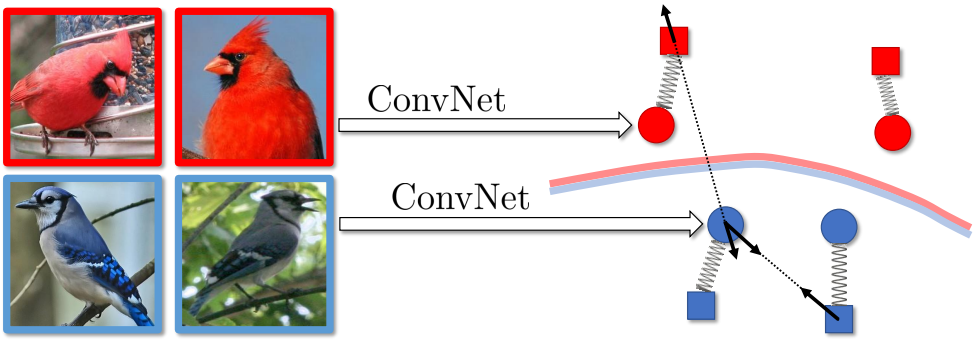


FIGURE 5.1: The schematic view of the "loose" impostor networks training. The trainable parameters include the CNN weights (that determine the mapping from training images to their embeddings shown as circles) and the set of impostor vectors (shown as squares). Each impostor possesses a class label, denoted by its color. Gradients of some loss terms are shown with arrows. They include the attraction term between an embedding of a training image and an impostor of the same class, as well as the repel term between an embedding of a training image and an impostor of another class. Additionally, loss terms penalize the deviation between the corresponding impostors and embeddings (shown with strings). At test time, the non-linear decision boundary of the classifier is determined by the position of impostors of various classes.

where σ is the standard deviation of the used kernel, which serves as a meta-parameter of the model. The value of σ is set by validation (although it can be included into the gradient-based learning formulation). The classification process then predicts the probability of the image $\mathbf{x}$ belonging to a certain class l using the radial basis function prediction rule:

$$p(l(\mathbf{x}) = l) = \frac{1}{\sum_{j=1}^{M} g(\mathbf{y}, \mathbf{c}_j; \sigma)} \sum_{j=1}^{M} g(\mathbf{y}, \mathbf{c}_j; \sigma) \left[l_j = l\right] , \tag{5.3}$$

where $[l_j = l]$ is an Iverson bracket.

The parameters of the embedding network and the reference set are obtained from training data given in the form of training examples $\mathbf{x}_1, \ldots, \mathbf{x}_M$ with labels $l_1, \ldots, l_M$. We introduce the same number of reference points $\mathbf{c}_1, \ldots, \mathbf{c}_M$, and associate each training example $\mathbf{x}_j$ with the reference point $\mathbf{c}_j$ (the particular ways of such association are discussed below). The label l_j is retained by the reference point $\mathbf{c}_j$, and thus is used after the training to classify new examples according to the RBF classification rule (5.3). Since each $\mathbf{c}_i$ serves as a certain representative of the training example $\mathbf{x}_i$ at test time, it is called $\mathbf{c}_i$ an *impostor* and the resulting architecture is called an *impostor network*.

5.1.1 Motivation

The estimation of class probabilities using the rule (5.3) can be performed efficiently even for a large number M given that the bandwidth σ is sufficiently small. Under

this condition, the approximate nearest neighbor search can be used to retrieve an (approximate) set of close neighbors of the embedding vector $\mathbf{y}$, for which the weights $g(\mathbf{y}, \mathbf{c}_j; \sigma)$ are not very close to zero. For a reasonably small M (in practice up to several tens of thousands, as in initial experiments), even an exhaustive computation of all weights in (5.3) constitutes a small fraction of the computation time, whereas the majority of the inference time is spent on the CNN computation in (5.1).

Compared to a standard CNN classifier, which utilizes linear classification on top of the feature hierarchy, the convolutional network inside the proposed architecture is used in conjunction with a highly non-linear RBF-classifier (5.3). Consequently, given a reasonable set of impostors, many fewer convolutional layers may be needed for the embedding f_θ in order to fit the resulting non-linear decision boundary defined by (5.3). This explains why in the experiments below impostor networks are able to achieve higher classification accuracy compared to CNNs with similar architecture.

5.1.2 Training impostor networks

There are several possible approaches how the parameters θ of the CNN f_θ and the impostor set $\mathbf{c}_1, \ldots, \mathbf{c}_M$ can be learned from a set of training examples $\mathbf{x}_1, \ldots, \mathbf{x}_M$.

Tied impostors. In the approach based on tied impostors, the learning results in $c_i = f_\theta(\mathbf{x}_i)$, i.e. each impostor is tied to the embedding of the training example x_i. The learning process can then be performed by minimizing the classification loss first introduced in [Goldberger et al., 2005]:

$$L(\theta, \mathbf{c}_1, \ldots, \mathbf{c}_M) = \quad -\tfrac{1}{M} \sum_{i=1}^{M} \log \frac{\sum_{j \neq i} g\Big(f_\theta(\mathbf{x}_i), \mathbf{c}_j; \sigma\Big)^{[l_j = l_i]}}{\sum_{j \neq i} g\Big(f_\theta(\mathbf{x}_i), \mathbf{c}_j; \sigma\Big)} \tag{5.4}$$

$$\text{subject to} \qquad \mathbf{c}_j = f_\theta(\mathbf{x}_j) \tag{5.5}$$

Each term in (5.4) approximates the probability estimate (5.3) for the correct class label of a training example (where the corresponding impostor is bypassed to avoid severe overfitting). The learning formulation (5.4-5.5) is, in fact, very similar to the one proposed in [Meyer et al., 2017] (and can also be regarded as an approach for deep metric learning). In practice, enforcing the hard constraint (5.5) during the optimization is non-trivial. Indeed, to evaluate the terms in (5.4) for a minibatch requires one requires to compute embeddings $\mathbf{c}_j = f_\theta(\mathbf{x}_j)$ for all close neighbors of the minibatch members. To address this challenge, [Meyer et al., 2017] suggested to maintain the cached copies (effectively, the impostors) of the embeddings at all times, that gradually become obsolete, as the optimization proceeds. The cached copies are updated by resetting to the actual embeddings once in several epochs. Such resetting leads to some problems, as it

effectively leads to abrupt and considerable change of the optimization objective, and may result in learning instabilities.

Fixed impostors. A simpler alternative, which in our experiments proved surprisingly efficient, is to fix the impostors at the beginning of the optimization process and never to update them. Thus, given the initial CNN parameters θ_0, every impostor $\mathbf{c}_i$ is initialized to $f_{\theta_0}(\mathbf{x}_i)$, and then the optimization of the objective (5.4) is performed over θ, while $\mathbf{c}_i$ are excluded from the optimization, and the constraint (5.5) is disregarded.

In our experiments, the initial state of the network parameters θ_0 corresponds to the result of the training on the well-known ILSVRC classification task. Given a good initialization θ_0, the initialized impostor set $\mathbf{c}_1, \dots \mathbf{c}_M$ are likely to create reasonable decision boundaries in the high-dimensional space $\mathbf{Y}$, which may be simpler to fit for the CNN f_θ than to achieve linear separability of the classes. This explains why such a simple scheme, when initialized to a pretrained θ_0, can outperform the standard CNN (starting with the same pre-initialization) considerably.

Loose impostors. The third scheme that can be seen as a generalization of the tied impostor scheme, which avoids its pitfalls. Here the impostor set is once again made a part of the optimization, however the hard constraint (5.5) is replaced with the soft penalty that drives the deviation between $\mathbf{c}_i$ and $f_\theta(\mathbf{x}_i)$ down, leading to the following learning formulation:

$$L(\theta, \mathbf{c}_1, .., \mathbf{c}_M) = \frac{1}{M} \sum_{i=1}^{M} \left(\lambda \left\| f_\theta(x_i) - \mathbf{c}_i \right\|^2 - \log \frac{\sum\limits_{j \neq i} g\left(f_\theta(\mathbf{x}_i), \mathbf{c}_j; \sigma \right) [l_j = l_i]}{\sum\limits_{j \neq i} g\left(f_\theta(\mathbf{x}_i), \mathbf{c}_j; \sigma \right)} \right). \quad (5.6)$$

Here, the parameter λ controls the relative weight of the attachment loss. We have found that the performance of the method is rather insensitive to λ and set it to 1 in our experiments. The loss (5.6) is differentiable w.r.t. all parameters of the impostor network, including both the CNN parameters and the impostor positions. One can therefore use standard stochastic gradient-based techniques to minimize it.

During a single training epoch, every impostor $\mathbf{c}_i$ participates in the classification of training samples multiple times. Each time, the partial gradient of the loss (5.6) with respect to $\mathbf{c}_i$ is computed and the impostor position $\mathbf{c}_i$ is updated accordingly (i.e. pulled towards the embeddings of the training examples of the same class or pushed away from the embeddings of the training examples of different classes). Once during every epoch, when the training example $\mathbf{x}_i$ is included in the mini-batch, the impostor $\mathbf{c}_i$ is also pulled towards the embedding $f_\theta(\mathbf{x}_i)$.

In the loose impostor formulation, untying the impostors from the embeddings of the training examples as well as from their initial approximations greatly increases the capacity of the model **without** increasing its computational complexity at test time. This is because the coordinates of impostors effectively become learnable parameters. This may both have a beneficial effect of decreasing underfitting and the negative effect of increasing overfitting. Also, compared to the tied impostors scheme, the gradients of the loss function in the loose scheme are computed without ignoring any terms and without cached approximations, making the learning process more stable. Compared to the fixed impostor scheme, the process can potentially improve the decision boundary (if the initialization θ_0 is highly sub-optimal). In the experimental section, we carefully compare all three impostor learning schemes.

5.2 Experiments

This section presents the experimental evaluation of the proposed impostor networks with various datasets and base CNN architectures.

Datasets. The experiments are performed on three fine-grained categorization datasets:

1. Caltech-UCSD Birds dataset (CUB-200-2011) [Wah et al., 2011], containing 11,788 bird images (5994 train and 5794 test images) of 200 classes.
2. Stanford Cars dataset [Krause et al., 2013], containing 16,185 car images (8144 train and 8041 test images) of 196 classes.
3. Landmarks-clean dataset [Gordo et al., 2017], containing 35423 images of 671 different landmarks (30837 train and 4586 test images). While usually used with landmark protocols, the task of landmark recognition may also be treated as a classification problem [Li et al., 2009].
4. Fungi 2018 dataset (part of 2018 version of iNaturalist Horn et al. [2017] contest) contains 85578 training images and 4182 validation images of 1394 fungi species.

For training the network all the datasets are split into train/validation/test subsets. The validation subset is used to tune the meta-parameters (including σ). The main performance measures are the standard classification accuracy and the inference time on both CPU and GPU.

Experimental details. During both training and applying the networks, the input images are resized to 256×256. When training, random cropping, and mirroring are

also applied. For the Cars and Birds datasets, we follow [Meyer et al., 2017] and use the provided bounding boxes to crop the object of interest in the image. Learning rates for all the schemes and the σ parameter of the RBF-classifier in the impostor networks were tuned on the validation subsets. For training, we use the Adam optimizer [Kingma and Ba, 2014] with the weight decay parameter equals 5×10^{-4}. All experiments were performed in the PyTorch [Paszke et al., 2017] framework.

The protocol for training the impostor networks includes the following steps:

1. The CNN is initialized with weights, obtained from the pretraining on ILSVRC. If the desired embedding dimensionality d_{em} does not equal to the dimensionality of the last fully-connected layer d_{fc}, the size of the last fully-connected layer is changed to be $d_{fc} \times d_{em}$, and the corresponding weights are initialized by a random matrix.

2. All the train images are passed through the CNN and the outputs of the last fully-connected layer are used as initial impostor vectors $c_1, \ldots, c_M$.

3. The impostors and the values of the matrix in the last layer of the network are divided by the average impostor $L2$-norm $\frac{1}{M} \sum_{i=1}^{M} ||c_i||$. This normalization trick is important in practice, as the typical scale of distances between impostors varies greatly with d_{em} and the scale of the last layer parameters. Due to this variability, the σ parameter has a very wide range of possible values, which makes it hard to tune. The division by the average norm allows the scale of distances to stay the same across different d_{em} and initializations.

4. The standard backpropagation is applied to minimize the corresponding impostor network loss: (5.5) or (5.6).

Due to a small number of images in the evaluation datasets, the RBF classification rule is computed with exhaustive search, i.e. distances to all impostors are calculated.

Compared approaches. The state-of-the-art methods for fine-grained recognition [Simon et al., 2017, Zheng et al., 2017] focus solely on the classification accuracy and it could take up to dozens of seconds to use them on mobile platforms. This work, however, is focused on targeting the demanding operating point of "lightweight" approaches. In particular, we consider the methods and network architectures that could be employed on non-GPU platforms and used in real-time, i.e. the inference should be faster than 10 FPS on CPU devices. Given these limitations, the following schemes are compared:

1. **CNN:** for this baseline the ILSVRC-pretrained network is fine-tuned on the particular dataset using the standard cross-entropy loss. The inference in this scheme is very efficient as it requires the only forward pass.

2. **CNN-extra:** to verify that simply adding more parameters into the CNN would not result in the accuracy boost, we compute this additional baseline, which mimics the CNN variant, except that an extra fully-connected layer with d_{em} neurons is inserted before the last layer

3. **ImpostorNet-tied:** the impostor network with impostors tied to the embeddings of the training examples. We follow [Meyer et al., 2017] and recompute the impostors every tenth epoch. Resetting impostors more frequently (e.g. after every epoch) was also tried but resulted in a worse performance.

4. **ImpostorNet-fixed:** the impostor network with impostors fixed to the initial positions determined by the ILSVRC-pretrained network.

5. **ImpostorNet-loose:** the impostor network with loose impostors that are part of the optimization process. Here, we stick to the value $\lambda = 1$ (in our initial experiments we observed that changing this parameter does not improve the final result).

All the networks in the compared schemes are initialized by the weights from the CNN, pretrained on the ILSVRC dataset [Russakovsky et al., 2015]. The train images' impostors are always initialized by the corresponding image embeddings, obtained with the initialization weights. For one of the three datasets, we have evaluated the following network architectures: SqueezeNet [Iandola et al., 2016] (PyTorch SqueezeNet version 1.1 was used), AlexNet [Krizhevsky et al., 2012], ResNet-18 and ResNet-50 [He et al., 2016]. As SqueezeNet stood out in terms of accuracy/efficiency trade-off in this comparison, this architecture was used for the remaining two datasets. Unless noted otherwise, the embedding dimensionality for ImpostorNets is set to 512.

Classification accuracy. The classification accuracy for all the compared methods is demonstrated in Table 5.1. Several key observations can be made from it:

1. For the Birds dataset, where different architectures are compared, the advantage of impostor networks is the most substantial for the most efficient Squeezenet architecture. For the more powerful architectures, the advantage of proposed scheme is much smaller. We believe that the reason for such behavior is that the architectures with a large number of parameters are able to make different classes linearly separable and the non-linear decision boundary is less useful.

TABLE 5.1: Classification accuracy on the Birds, Cars and Landmarks datasets for three versions of impostor networks and the CNN trained with cross-entropy loss (including the variant with additional learnable parameters). Impostor networks provide a substantial performance improvement on the Birds and Cars datasets in the case of SqueezeNet architecture.

	Birds				Cars	Landmarks
	SqueezeNet	AlexNet	ResNet-18	ResNet-50	SqueezeNet	SqueezeNet
CNN	70.72	65.58	79.61	82.35	78.65	92.48
CNN-extra	69.30	65.28	80.08	82.58	76.67	92.67
tied	70.04	62.10	75.68	80.54	80.84	93.65
fixed	75.47	66.97	80.05	82.68	79.87	93.02
loose	76.01	67.30	80.58	82.14	85.05	92.83

2. The version with loose impostors outperforms the baseline and versions with tied and fixed impostors in several cases and never performs much worse. We attribute this fact to the extra learning capacity of this scheme, explain it by the fact that in the "loose" version the impostors are trained jointly with the networks, hence many more model parameters adapt to the particular dataset.

3. The advantage of impostor nets cannot be explained simply by having extra parameters in the decision function. In fact, adding more parameters into the CNN does not necessarily improves the performance (CNN-extra vs. CNN).

4. The advantage of impostor nets on the Landmarks-clean dataset is small. This may be due to already saturated performance on this dataset.

Additional experiments were performed to explore ImpostorNets performance in different situations.

RBF-SVM baseline. To prove the necessity of joint learning of ConvNet weights and impostor positions in embedding space, we compare ImpostorNets with RBF-SVM classifier trained on fixed feature vectors extracted from the last hidden layer of ConvNet. This experiment was performed for Squeezenet architecture on Birds dataset and yielded the accuracy of 69.2%, lower than original ConvNet and every ImpostorNet version.

Fungi 2018. We have performed additional experiments on Fungi 2018 dataset to confirm the applicability of impostor networks for larger datasets. Impostor networks in the "loose" version with SqueezeNet architecture achieved an accuracy of 26.6%, compared with 25.7% of original ConvNet.

Dependence on the embedding dimensionality $\mathbf{d_{em}}$. The dimensionality of the embeddings and the impostors d_{em} has a large influence on the impostor networks performance. Figure 5.2 demonstrates the classification accuracy on the Birds dataset as a function of d_{em} for the "fixed" version of the impostor network. As expected, larger

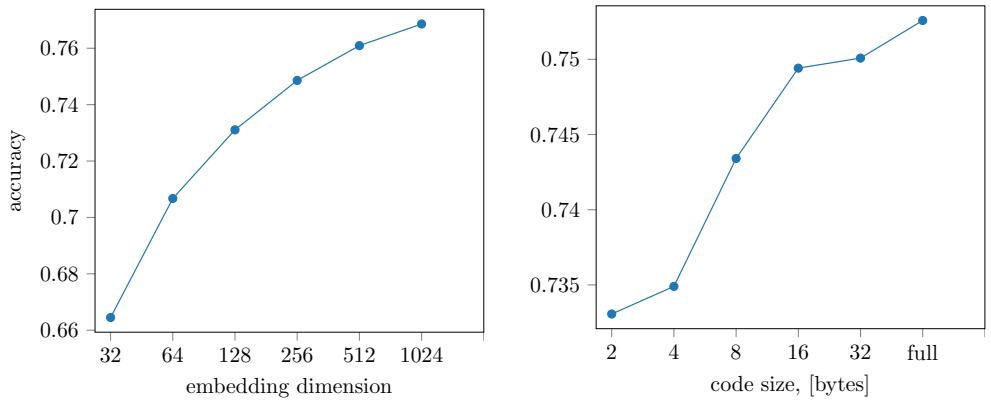


FIGURE 5.2: Left – the classification accuracy of "fixed" impostor networks on the Birds dataset as a function of the impostors dimensionality d_{em}. Larger d_{em} results in higher performance but also increases the memory consumption and the computational complexity of RBF classification (although the computational cost still remains very small compared to the network inference time). Right – same accuracy as a function of the compressed impostor representation size (which is given in bytes). The impostors could be compressed to as little as 16 or 32 bytes with negligible accuracy drop compared to uncompressed impostors.

dimensionalities result in more powerful models (due to larger number of parameters). On the other hand, the large d_{em} values increase the memory consumption and RBF classification complexity. In most of the experiments d_{em} is set to 512.

Initialization variants. In the "fixed" scheme the impostors are not updated during training and are fully defined by the initialization weights. In this experiment, we compare several different ways to initialize the "fixed" impostors with the SqueezeNet architecture. The initialization with the weights, obtained with pretraining on the ILSVRC dataset, results in 76% classification on the Birds dataset. As expected, the random initialization of impostors results in poor classification performance of 16.1%. Another reasonable way is to initialize the impostors with the weights obtained after finetuning on the Birds dataset with cross-entropy loss. This initialization results in slightly higher final performance of 76.9%. However, this increase in the accuracy comes at the cost of the additional CNN training.

Impostors compression. Here we investigate the performance of the "fixed" version of the proposed approach when impostors are represented by the compact PQ codes to reduce memory consumption. In particular, the optimized version of PQ (OPQ [Ge et al., 2014]) was chosen for compression. Figure 5.2–right demonstrates the classification accuracy on the Birds dataset as a function of the code size (presented in bytes). In this experiment, we initialize the impostors, compress them with OPQ, and then train the impostor network using the compressed vectors as the impostor set $c_1, \ldots, c_M$. The graph shows that the original 512-dimensional impostors could be compressed to 16 or 32 bytes with only negligible accuracy drop. Note, that the usage of PQ codes for impostors

does not prevent the efficient RBF classification, as the distances to the compressed impostors could be computed efficiently both on CPU and GPU [Johnson et al., 2017]. Overall, the total memory consumption of the impostor networks is dominated by the memory required to store CNN weights if the impostors are PQ-compressed. E.g. for the Birds dataset, the size of the SqueezeNet model is about 4.8MB, while the impostors, compressed to 16 bytes, require only 92KB of additional memory.

2D visualizations. Visualizing relation between impostors and embeddings, while desirable, is hard to because of embedding space have high dimensionality, and traditional visualization techniques such as t-SNE can introduce artifacts. However, we can train a separate toy model with two-dimensional embedding space. This toy model allows to directly observe impostors and embeddings, and some observations can be made.

The learning process for ImpostorNets trained on MNIST dataset is visualized on Figure 5.3. Both impostors and embedding start from the single pile in the central area at the first epoch, and the classification loss tries to scatter this pile, so both the impostors and embeddings start their travel outside. Starting from the epoch 2, it is clearly seen that embeddings lead and impostors follow.

The strength of bond between impostors and embeddings is controlled by parameter λ. The effect of varying λ on two-dimensional embedding space is presented on 5.4. Small λ allows embeddings to travel almost independently of impostors, similarly to the fixed impostors setting. Large lambda λ makes mismatch loss produce very large gradients and makes model rigid, leading to a bad fit.

5.2.1 Timings

In this section, the timings of ImpostorNets are measured both on CPU and GPU. The GPU timings are recorded on single Tesla K40m with CUDA 8.0, and for CPU timings, Intel Xeon CPU E5-2650 v2 2.60GHz was used.

Note, that the state-of-the-art approaches for fine-grained classification rarely take inference timings into consideration and do not report them. To position the proposed approach among the previous works, we also compute the timings for two recent approaches, which have their models publicly available [Simon et al., 2017, Zheng et al., 2017]. Figure 5.5 shows the timings, achieved by ImpostorNets as well as the timings of the state-of-the-art approaches. All the timing are computed on the Birds dataset.

The top part of Figure 5.5 demonstrates the timings of impostors networks with different architectures. The timings of CNN forward pass and distances computation are reported separately to demonstrate that the contribution of the latter is quite small for

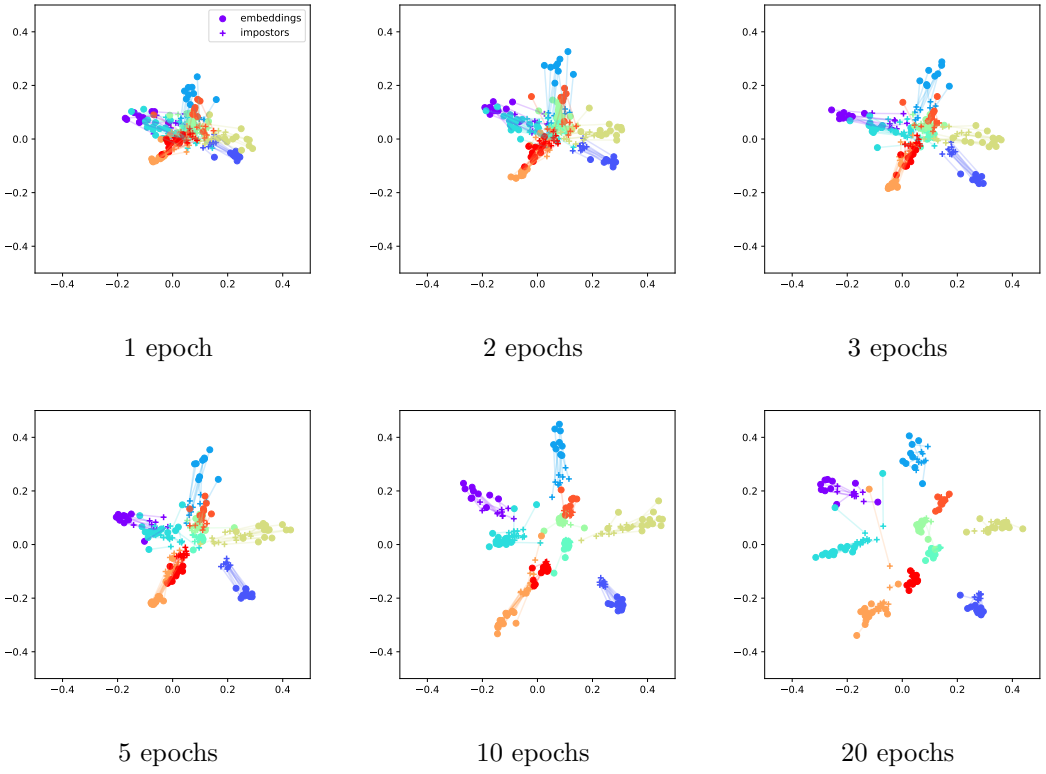


FIGURE 5.3: A learning process for ImpostorNets with 2D embedded space on MNIST digits dataset. Embeddings are designated by crosses and impostors by squares, corresponding impostors and embeddings are connected by straight line, and classification label is encoded by color. 1 in 500 of samples from the dataset is drawn to avoid clutter. In the middle of the learaning process impostors are dragged behind embeddings.

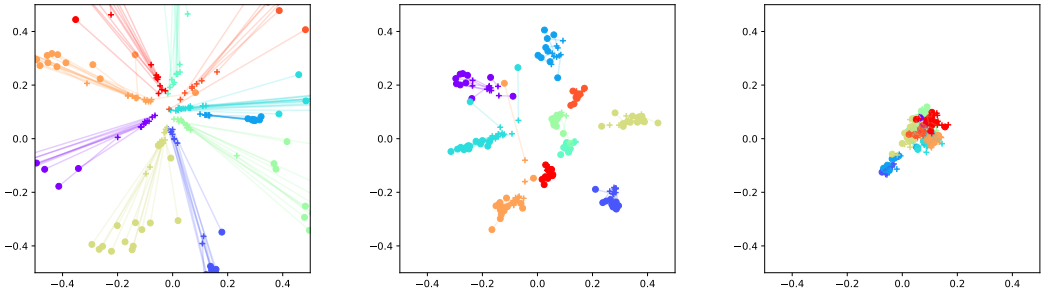


FIGURE 5.4: The influence of λ parameter on the distribution of 2D embeddings and impostors for MNIST digits dataset achived after fixed number of iterations. Small lambda on the left allows embeddings to travel large distance almost separately from impostors, similarly to the fixed impostors scenario. On the right, large lambda makes optimization difficult and ImpostorNet stucks in the position close to the initial state. Central panel corresponds to optimal scenario, in which both impostors and embeddings move.

all architectures, even with exhaustive nearest neighbor search. Of course, for larger training sets exhaustive search could be inefficient and the approximate methods should be used.

The bottom parts of Figure 5.5 compares the "loose" version of ImpostorNets with the existing approaches in terms of time-accuracy trade-off on the Birds dataset. The black points correspond to the performance of two recent methods, $\alpha-$pooling [Simon et al., 2017] and multiple attention with different parameters (MA-ConvNet-2 and MA-ConvNet-4) [Zheng et al., 2017]. The blue points correspond to the performance of CNN trained with the standard cross-entropy loss. Finally, the orange points denote the performance of the proposed system (loose impostors).

In general, the proposed impostor networks improve the classification accuracy with a negligible increase in the computational cost. The improvement is most noticeable for the "lightweight" SqueezeNet architecture, what makes the proposed system an excellent choice for mobile platforms. Note, the state-of-the-art methods [Simon et al., 2017, Zheng et al., 2017] achieve higher classification accuracy, but their runtimes are orders of magnitude slower on both CPU and GPU, which restricts its usage in practice.

5.2.2 Open set recognition.

This experiment demonstrates that the proposed impostor networks could be successfully used in the "open set learning" scenario when the images of the classes unseen during training could be submitted to a network at test time. The ability to detect the inputs of unknown classes is an important practical property for many application scenarios of fine-grained recognition. Here, we investigate the ability to detect unknown classes based on the confidence (entropy) of network predictions.

It is well-known that the CNNs trained with the standard cross-entropy loss tend to be overconfident even when they are wrong [Guo et al., 2017]. Interestingly, the proposed impostor networks are able to express uncertainty due to the usage of RBF classification at the final step. To support this claim, the following experiment is performed. we train the cross-entropy CNN and the three variants of the impostor networks with the same SqueezeNet architecture on the Birds dataset. The models are then evaluated on two test sets: Test1 set containing birds images and Test2 set containing the non-bird images from the ILSVRC dataset. For each of these sets, we pass the images through the networks and compute the entropy of the class labels probability distribution. The histograms of the entropy values are presented on Figure 5.6. For the standard CNN, two distributions are very close, which means that the degree of the network confidence

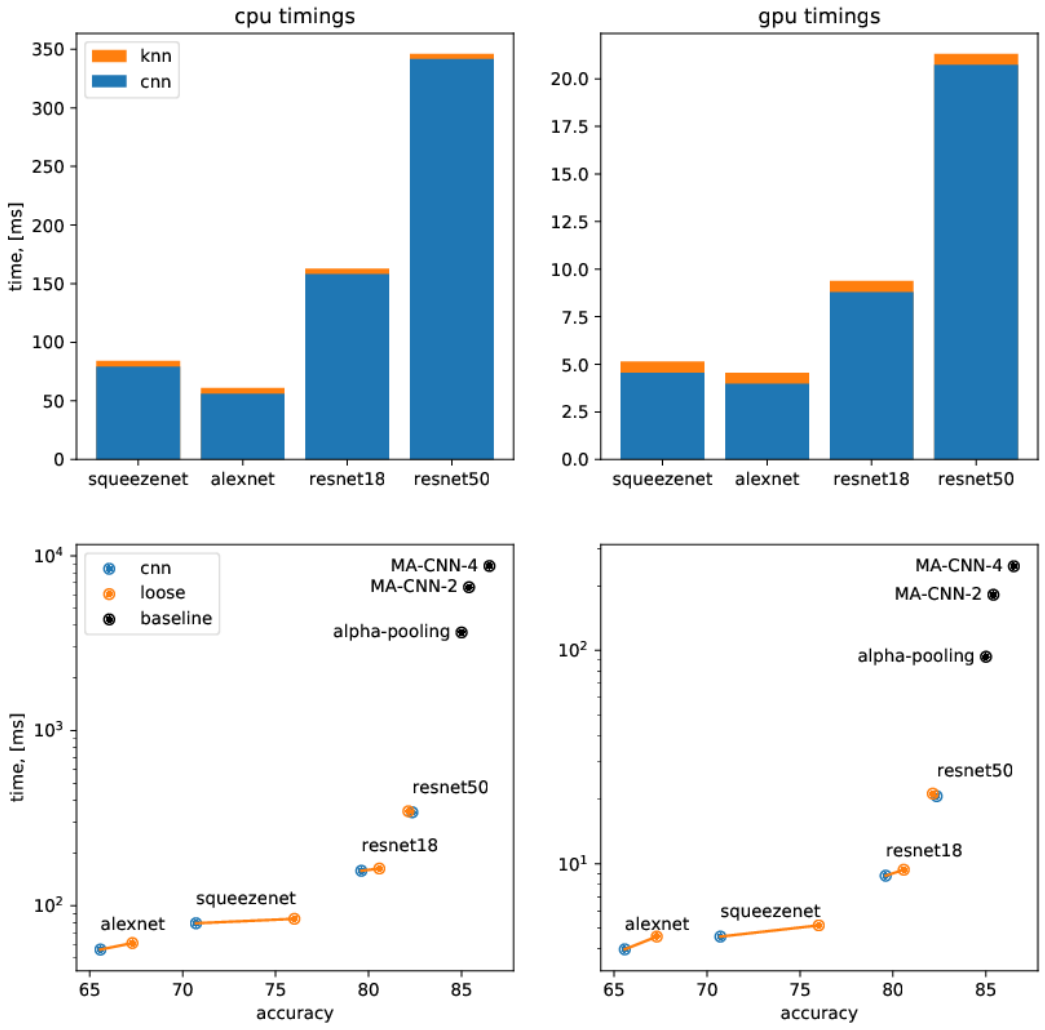


FIGURE 5.5: Top panels: the CPU and GPU (Tesla K40) timings for the impostor networks with different CNN architectures on the Birds dataset. Blue bars correspond to the computational times of CNN forward pass, while orange bars correspond to the RBF classification rule computation. The RBF contribution into the total runtime is negligible for all architectures. Bottom panels: comparison of the "loose" impostor networks with the state-of-the-art approaches in terms of the runtime-accuracy trade-off on the Birds dataset. The proposed impostor networks are orders of magnitude faster the state-of-the-art methods with a decent decrease in the classification accuracy.

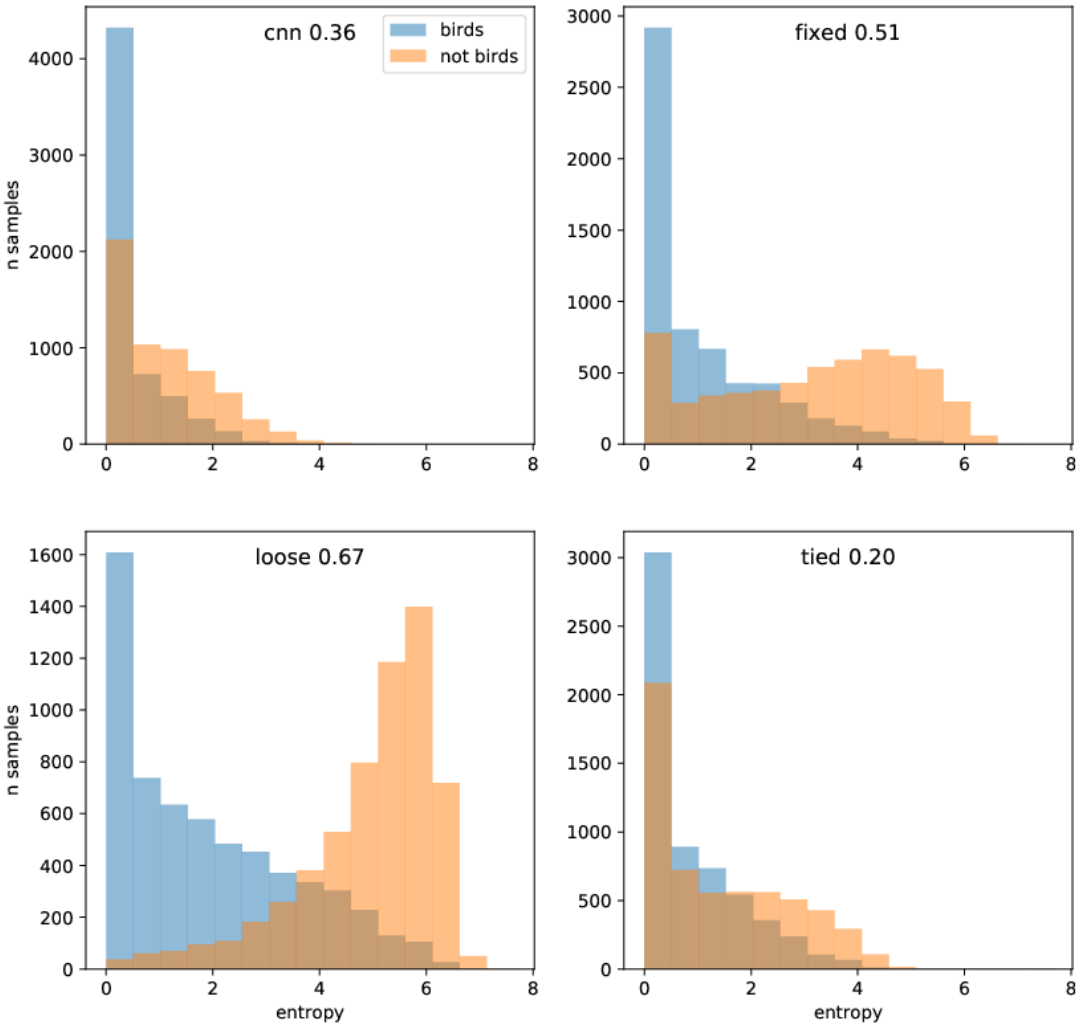
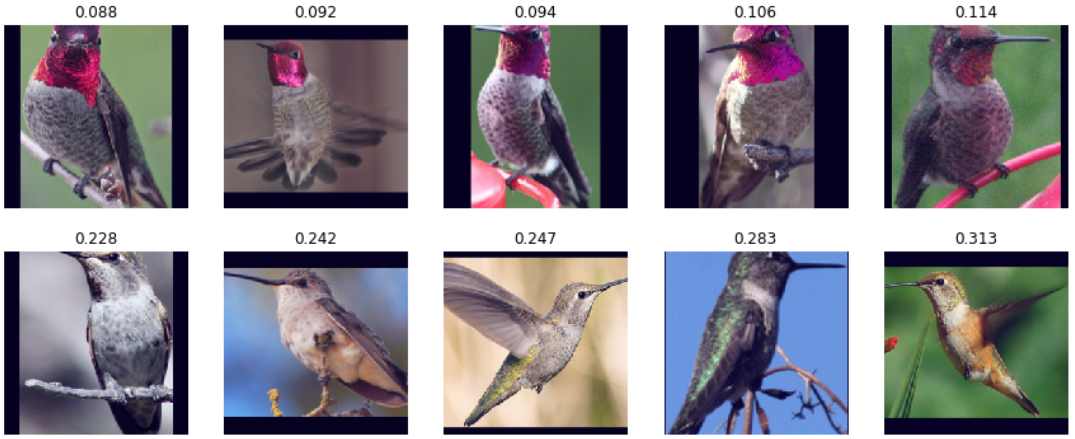


FIGURE 5.6: Histograms of entropy values of the probability distributions obtained with the cross-entropy CNN and the impostor networks with the Squeezenet architecture. All the networks were trained on the Birds dataset and applied to the sets of bird and non-bird images. The cross-entropy CNN is almost equally confident on the images of both seen and unseen classes. The "fixed" and "loose" impostor networks are able to detect their uncertainty based on much higher entropy values on the non-bird images. The number on each plot corresponds to the Kolmogorov-Smirnov distance between the distributions of the entropy values.

is the same for seen and unseen classes. The fixed and loose impostor networks clearly tend to be less confident in their predictions for the images of unseen classes.

5.2.3 Intuition behind the "loose" impostors

Finally, we perform an experiment that allows obtaining an intuition about mutual positions of the image embeddings and the corresponding "loose" impostors in the embedding space. This experiment uses the training images of Anna's hummingbird class



(A) Anna's hummingbird

(B) tanager

FIGURE 5.7: Examples from two classes of the Birds dataset: Anna's hummingbird (top panel) and scarlet tanager (bottom panel). On both panels, samples with the smallest distances between their embeddings and the corresponding impostors (loose scheme) are shown in the top row, and samples with the largest distances are in the bottom row. The largest distances correspond to samples that are generally hard to classify. Anna's hummingbirds exhibit pronounced sexual dimorphism: males have the characteristic bright magenta crown, while females are much bleaker. Indeed, all the specimen on the top row are males and on the bottom row — females. For scarlet tanager, the classification is complicated then its black wings are not clearly seen, as they are the defining feature which separates scarlet tanager from a similar class of summer tanager.

of the Birds dataset and computes the distances between their embeddings and the corresponding impostors. The bottom row of Figure 5.7 demonstrates top-5 images with the largest embedding-impostor distances. These images are hard to classify as they exclusively contain female hummingbirds, which lack the distinctive coloring of male Anna's hummingbirds. The top row of Figure 5.7 visualizes top-5 images with the smallest embedding-impostor distances. These images correspond to easy samples of brightly colored male hummingbirds, with embeddings which are far away from the decision boundary in the embedding space.

5.3 Conclusion

This chapter describes a new framework of impostor networks that consist of the deep convolutional network with a non-parametric classifier on top. The CNN and the RBF parts are learned jointly, and we explore three possible ways to perform such joint learning. While the RBF part could benefit the CNN of any architecture but the most significant accuracy boost is achieved for the efficient SqueezeNet architecture what makes the impostor networks a good choice for resource-constrained settings, e.g. mobile platforms.

On top of an accuracy increase for compact architectures, impostor networks exhibit several others advantageous properties, including robustness in open-set scenario and possibility to adjust training data without complete retraining of CNN part. Detailed exploration of these characteristics could transform impostor nets into a useful general purpose method for pattern recognition, but this topic lies outside of the scope of this thesis.

Chapter 6

Conclusion and Discussion

The goal of this thesis is to address the problem of low execution speed, associated with modern convolutional neural networks and explore different approaches to solve this problem. The direct comparison of proposed approaches is complicated by the hardware change and the evolution of deep learning frameworks which occurred during our work on the contents of this thesis (the importance of these factors is demonstrated in Chapter 1). Nevertheless, in this section, we summarize the contents of the thesis, provide some comparisons and define the area of applicability for each method.

In Chapter 2 we have provided a systematic review of the literature on the topic of speeding up convolutional neural networks. The approaches are divided into several groups: tensor decompositions, quantization, pruning, teacher-student, manual and automatic search for efficient architectures, and adaptive models.

In Chapter 3 we have proposed a method for speeding up convolutions in neural networks with low-rank CP-decomposition of convolutional weights. The method implementation is based on existing building blocks of CNNs which allows for easy deployment, and most importantly, finetuning of the model, although the instability of CP-decomposition complicates the finetuning process. The experimental results demonstrate impressive speed ups with minimal accuracy drops for several architectures.

The main limiting factor of CP-decomposition method in the form tested in this thesis is the layer-wise application, which limits its performance for deeper networks. On the other hand, the explicit decomposition along spatial dimensions is effective for the large filters. Thus, the CP-decomposition method is the most effective for shallow networks with large filters, as demonstrated by the experiments on the character recognition task. The relevance of this method decreases as the modern architectures are becoming deeper and deeper, and the filters larger then 3×3 are rarely used.

Applied to the convolutional layer with C input and N output channels and $d \times d$ filters, which uses NCd^2 mult-add operations per pixel, decomposed convolution requires $R(C+2d+N)$ operations per pixel, where R is the rank of decomposition. The speedup ratio $\frac{NCd^2}{R(C+2d+N)}$ depends on the parameters of the layer, as well as on the rank R, which could not be estimated in advance. Thus, the estimation of the performance of this method requires experimentation.

Chapter 4 describes a novel method for speeding up CNN through pruning. We demonstrate that group sparsity regularizer embedded into stochastic gradient descent minimization can accomplish group-wise brain damage efficiently. The experiments show that a carefully designed group-wise brain damage procedure can sparsify existing neural networks considerably. It is demonstrated what efficient neural networks may operate non-rectangular filters of different shapes.

The main advantage of brain damage method over CP-decomposition is in the fact that all the layers can be pruned simultaneously. This property allows the method to obtain competitive results for the deepest architectures and stay relevant, as demonstrated by the continuing streak of publications with the variations of pruning techniques. The performance of this method is not tied to the filter size.

Sparse convolution requires τNCd^2 operations, where τ is the sparsity level. Similarly to the case of CP-decomposition, the minimal achievable value of sparsity level τ cannot be estimated in advance. The empirical comparison on the case of acceleration of a single layer of AlexNet architecture demonstrates the advantage of brain damage method. Although 5×5 filters of this layer are not very large, most of the modern networks rely on even smaller 3×3 layers.

In Chapter 5, a new framework of impostor networks is proposed for efficient fine-grained classification. The impostor networks consist of the deep convolutional network with a non-parametric classifier on top. The CNN and the RBF parts are learned jointly, and we investigate three possible ways to perform such joint learning.

This approach does not modify the structure of convolutional layers, and the speedup comes from the possibility to solve the same task with lighter architecture. This approach could benefit any CNN, but the most significant advantage is expected from light architectures, lacking the capacity to achieve a complete linear separability on the target dataset. This theoretical consideration is confirmed by the experiments demonstrating that the maximal advantage is achieved for the compact SqueezeNet architecture, making the impostor networks an excellent choice for resource-constrained settings, such as smartphones and wearable devices.

With so many different directions of speeding up neural networks, it is crucial to determine the most promising ones regarding both practical application and future research. Our ability to predict future depends on the level of maturity of the research topic. Tensor decompositions stand out as a particularly well-developed field, with several methods ready for practical applications, established influence on other approaches and little expectancy for groundbreaking discoveries. Binarization of neural networks also has been studied for some time, but the separation between research and practical application in this area still exists. This situation could drastically change in the future with the development of the new kinds of hardware architectures.

The single most promising approach in the field is probably an automatic architecture search, as it has a potential to absorb and combine the advantages of other methods, including tensor decompositions, all kinds of quantization, pruning and teaches-student approaches. Although initial work on the topic [Zoph and Le, 2017] is known for using computational resources which are unattainable for most researchers, the situation is changing for the better.

One may also notice what high computation cost is inseparably tied to the very idea of deep learning, which is to stack multiple layers of linear operations. Thus, some speed barrier exists, and to go below it, we have to switch to an entirely new kind of models. At this point, convolutional neural networks are so ubiquitous in computer vision, and their capabilities are so beyond of other algorithms, that it is hard to imagine their replacement by something else. Possibly further development of teacher-student approaches will allow transferring the capabilities of CNNs onto much faster models, such as trees.

The basic approach explored in this thesis is top-to-bottom: we try to bring the execution time of an existing neural network down. The task to build fast visual recognition algorithm can be approached from the opposite direction, in the bottom-to-top fashion. Such an approach starts with extremely fast models [Kumar et al., 2017, Gupta et al., 2017, Garg et al., 2018] which can currently solve only relatively simple tasks and tries to expand them for visual recognition. While it is not presently clear if it is possible, the advances in this direction have the potential to change the landscape for fast methods for computer vision completely.

Bibliography

Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frédéric Pétrot. Ternary neural networks for resource-efficient AI applications. In *International Joint Conference on Neural Networks*, 2017.

Genevera Allen. Sparse higher-order principal components analysis. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2012.

Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Fixed point optimization of deep convolutional neural networks for object recognition. In *Acoustics, Speech, and Signal Processing (ICASSP), International Conference on*, 2015.

M. Astrid and Seung-Ik Lee. Cp-decomposition with tensor power method for convolutional neural networks compression. In *Big Data and Smart Computing*, 2017.

Hossein Azizpour, Ali Sharif Razavian, Josephine Sullivan, Atsuto Maki, and Stefan Carlsson. From generic to specific deep representations for visual recognition. In *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2015.

Artem Babenko and Victor Lempitsky. Aggregating local deep features for image retrieval. In *International Conference on Computer Vision (ICCV)*, 2015.

Hessam Bagherinezhad, Mohammad Rastegari, and Ali Farhadi. LCNN: Lookup-based convolutional neural network. *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2017.

Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Practical neural network performance prediction for early stopping. *arXiv preprint arXiv:1705.10823*, 2017.

L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, and Greg Henry. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 2002.

Tolga Bolukbasi, Joseph Wang, Ofer Dekel, and Venkatesh Saligrama. Adaptive neural networks for efficient inference. In *International Conference on Machine Learning (ICML)*, 2017.

Cristian Bucila, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2006.

Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Reinforcement learning for architecture search by network transformation. *arXiv preprint arXiv:1707.04873*, 2017.

Gal Chechik, Isaac Meilijson, and Eytan Ruppin. Synaptic pruning in development: a computational account. *Neural computation*, 1998.

Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*, 2006.

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Francois Chollet. Xception: Deep learning with depthwise separable convolutions. *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2017.

Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In *International conference on artificial neural networks*, 2014.

Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The Cityscapes dataset for semantic urban scene understanding. In *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2016.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training deep neural networks with binary weights during propagations. *Advances in Neural Information Processing Systems (NIPS)*, 2015.

Mark Craven and Jude W Shavlik. Extracting tree-structured representations of trained networks. In *Advances in Neural Information Processing Systems (NIPS)*, 1996.

Vin De Silva and Lek-Heng Lim. Tensor rank and the ill-posedness of the best low-rank approximation problem. *SIAM J. Matrix Anal. Appl.*, 2008.

Emily Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. *arXiv preprint arXiv:1404.0736*, 2014.

Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. DeCAF: A deep convolutional activation feature for generic visual recognition. In *International Conference on Machine Learning (ICML)*, 2014.

Michael Figurnov, Maxwell D Collins, Yukun Zhu, Li Zhang, Jonathan Huang, Dmitry Vetrov, and Ruslan Salakhutdinov. Spatially adaptive computation time for residual networks. *arXiv preprint arXiv:1612.02297*, 2017.

Mikhail Figurnov, Aizhan Ibraimova, Dmitry P Vetrov, and Pushmeet Kohli. PerforatedCNNs: Acceleration through elimination of redundant convolutions. In *Advances in Neural Information Processing Systems (NIPS)*, 2016.

I. Freeman, L. Roese-Koerner, and A. Kummert. EffNet: An Efficient Structure for Convolutional Neural Networks. *arXiv preprint arXiv:1801.06434*, 2018.

Nicholas Frosst and Geoffrey Hinton. Distilling a neural network into a soft decision tree. *arXiv preprint arXiv:1711.09784*, 2017.

Kunihiko Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. *Competition and cooperation in neural nets*, 1982.

Vikas K. Garg, Ofer Dekel, and Lin Xiao. Learning small predictors. *arXiv preprint arXiv:1803.02388*, 2018.

Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576*, 2015.

Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014.

Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? The KITTI vision benchmark suite. In *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2012.

Ross Girshick. Fast R-CNN. In *International Conference on Computer Vision (ICCV)*, 2015.

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2014.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010.

Jacob Goldberger, Geoffrey E Hinton, Sam T. Roweis, and Ruslan R Salakhutdinov. Neighbourhood components analysis. In *Advances in Neural Information Processing Systems (NIPS)*, 2005.

Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

Albert Gordo, Jon Almazán, Jerome Revaud, and Diane" Larlus. Deep image retrieval: Learning global representations for image search. In *European Conference on Computer Vision (ECCV)*, 2017.

Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Tien-Ju Yang, and Edward Choi. Morphnet: Fast & simple resource-constrained structure learning of deep networks. *arXiv preprint arXiv:1711.06798*, 2017.

Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On calibration of modern neural networks. *International Conference on Machine Learning (ICML)*, 2017.

Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. ProtoNN: Compressed and accurate kNN for resource-scarce devices. In *International Conference on Machine Learning (ICML)*, 2017.

Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems (NIPS)*, 2015.

Kaiming He and Jian Sun. Convolutional neural networks at constrained time cost. In *Conference on Computer Vision and Pattern Recogonition (CVPR)*, June 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2016.

Xiangteng He and Yuxin Peng. Fine-grained image classification via combining vision and language. In *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2017.

Geoffrey Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

Geoffrey E Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *NIPS Deep Learning Workshop*, 2014.

Frank L Hitchcock. The expression of a tensor or a polyadic as a sum of products. *Journal of Mathematics and Physics*, 1927.

Grant Van Horn, Oisin Mac Aodha, Yang Song, Alexander Shepard, Hartwig Adam, Pietro Perona, and Serge J. Belongie. The inaturalist challenge 2017 dataset. *arXiv preprint arXiv:1707.06642*, 2017.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*, 2016.

Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *European Conference on Computer Vision (ECCV)*, 2016.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2017.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. *Advances in Neural Information Processing Systems (NIPS)*, 2016a.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061*, 2016b.

Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv preprint arXiv:1602.07360*, 2016.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Deep features for text spotting. In *European Conference on Computer Vision (ECCV)*, 2014a.

Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. In *British Machine Vision Conference (BMVC)*, 2014b.

Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Hamming embedding and weak geometric consistency for large scale image search. In *European Conference on Computer Vision (ECCV)*, 2008.

Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2011.

Rodolphe Jenatton, Jean-Yves Audibert, and Francis Bach. Structured variable selection with sparsity-inducing norms. *The Journal of Machine Learning Research*, 2011.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *ACM Multimedia*, 2014a.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014b.

Jonghoon Jin, Aysegul Dundar, and Eugenio Culurciello. Flattened convolutional neural networks for feedforward acceleration. *arXiv preprint arXiv:1412.5474*, 2014.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *arXiv preprint arXiv:1702.08734*, 2017.

Justin Johnson, Alexandre Alahi, and Fei-Fei Li. Perceptual losses for real-time style transfer and super-resolution. *International Conference on Machine Learning (ICML)*, 2016.

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon,

James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations (ICLR)*, 2014.

Eleftherios Kofidis and Phillip A Regalia. On the best rank-1 approximation of higher-order supersymmetric tensors. *SIAM J. Matrix Anal. Appl.*, 2002.

T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Rev.*, 2009.

Shu Kong and Charless C. Fowlkes. Low-rank bilinear pooling for fine-grained classification. In *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2017.

Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3D object representations for fine-grained categorization. In *4th International IEEE Workshop on 3D Representation and Recognition*, 2013.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2012.

Ashish Kumar, Saurabh Goyal, and Manik Varma. Resource-efficient machine learning in 2 KB RAM for the internet of things. In *International Conference on Machine Learning (ICML)*, 2017.

Andrew Lavin. Fast algorithms for convolutional neural networks. *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2016.

Vadim Lebedev and Victor Lempitsky. Fast ConvNets using group-wise brain damage. In *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2016.

Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan V. Oseledets, and Victor S. Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *International Conference on Learning Representations (ICLR)*, 2015.

Vadim Lebedev, Artem Babenko, and Victor Lempitsky. Impostor networks for fast fine-grained recognition. *arXiv preprint arXiv:1806.05217*, 2018.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1989.

Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems (NIPS)*, 1990.

Jinyu Li, Rui Zhao, Jui-Ting Huang, and Yifan Gong. Learning small-size DNN with output-distribution-based criteria. In *Interspeech*, 2014.

Yunpeng Li, David J. Crandall, and Daniel P. Huttenlocher. Landmark classification in large-scale image collections. In *International Conference on Computer Vision (ICCV)*, 2009.

Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *International Conference on Learning Representations (ICLR)*, 2014.

Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. *International Conference on Learning Representations (ICLR)*, 2016.

Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2015a.

Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017.

Lingqiao Liu, Chunhua Shen, and Anton van den Hengel. The treasure beneath convolutional layers: Cross-convolutional-layer pooling for image classification. In *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2015b.

Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2017.

Michael Mathieu, Mikael Henaff, and Yann Lecun. Fast training of convolutional networks through FFTs. In *ICLR2014*, 2014.

Benjamin J. Meyer, Ben Harwood, and Tom Drummond. Nearest neighbour radial basis function solvers for deep neural networks. *arXiv preprint arXiv:1705.09780*, 2017.

Asit K. Mishra, Eriko Nurvitadhi, Jeffrey J. Cook, and Debbie Marr. WRPN: wide reduced-precision networks. *arXiv preprint arXiv:1709.01134*, 2017.

Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *arXiv preprint arXiv:1611.06440*, 2016.

NervanaSystems. NervanaGPU. https://github.com/NervanaSystems/nervanagpu, 2015.

Behnam Neyshabur, Ryota Tomioka, Ruslan Salakhutdinov, and Nathan Srebro. Data-dependent path normalization in neural networks. *arXiv preprint arXiv:/1511.06747*, 2015.

Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry Vetrov. Tensorizing neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2015.

Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2015.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. *International Conference on Learning Representations (ICLR)*, 2017.

H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

Anh-Huy Phan, Petr Tichavsky, and Andrzej Cichocki. Low complexity damped gauss–newton algorithms for candecomp/parafac. *SIAM Journal on Matrix Analysis and Applications*, 34(1):126–147, 2013.

James Philbin, Ondrej Chum, Michael Isard, Josef Sivic, and Andrew Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2007.

Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. *International Conference on Learning Representations (ICLR)*, 2018.

Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *International Conference on Machine Learning (ICML)*, 2009.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision (ECCV)*, 2016.

E. Real, A. Aggarwal, Y. Huang, and Q. V Le. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*, 2018.

Joseph Redmon and Ali Farhadi. YOLO9000: Better, faster, stronger. *arXiv preprint arXiv:1612.08242*, 2016.

Joseph Redmon and Ali Farhadi. YOLOv3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2016.

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2015.

Roberto Rigamonti, Amos Sironi, Vincent Lepetit, and Pascal Fua. Learning separable filters. In *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2013.

Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. *International Conference on Learning Representations (ICLR)*, 2015.

Volker Roth and Bernd Fischer. The group-lasso for generalized linear models: uniqueness of solutions and efficient algorithms. In *International Conference on Machine Learning (ICML)*, 2008.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 2015.

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 2016.

Richard Shin, Charles Packer, and Dawn Song. Differentiable neural network architecture search. In *International Conference on Learning Representations (ICLR)*, 2018.

Marcel Simon, Yang Gao, Trevor Darrell, Joachim Denzler, and Erik Rodner. Generalized orderless pooling performs implicit salient matching. *International Conference on Computer Vision (ICCV)*, 2017.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *International Conference on Learning Representations (ICLR)*, 2015.

L. Sorber, M. Van Barel, and L. De Lathauwer. Tensorlab v2.0. http://tensorlab.net, 2014.

Alwin Stegeman and Pierre Comon. Subtracting a best rank-1 approximation may increase tensor rank. *Linear Algebra Appl.*, 2010.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, et al. Going deeper with convolutions. In *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2015.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2016.

Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections. *Association for the Advancement of Artificial Intelligence*, 2017.

Surat Teerapittayanon, Bradley McDanel, and HT Kung. BranchyNet: Fast inference via early exiting from deep neural networks. In *International Conference on Pattern Recognition (ICPR)*, 2016.

Sebastian Thrun. Extracting rules from artificial neural networks with distributed representations. In *Advances in Neural Information Processing Systems (NIPS)*, 1995.

Giorgio Tomasi and Rasmus Bro. A comparison of algorithms for fitting the PARAFAC model. *Comp. Stat. Data An.*, 2006.

Antonio Torralba, Rob Fergus, and William T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2008.

Dmitry Ulyanov, Vadim Lebedev, Andrea Vedaldi, and Victor Lempitsky. Texture networks: Feed-forward synthesis of textures and stylized images. *International Conference on Machine Learning (ICML)*, 2016.

Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on CPUs. In *NIPS Deep Learning and Unsupervised Feature Learning Workshop*, 2011.

A. Vedaldi and K. Lenc. MatConvNet – convolutional neural networks for matlab. *arXiv preprint arXiv:1412.4564*, 2014.

C. Wah, S. Branson, P. Welinder, P. Perona, and S. Belongie. The Caltech-UCSD Birds-200-2011 Dataset. Technical report, California Institute of Technology, 2011.

Peisong Wang and Jian Cheng. Accelerating convolutional neural networks for mobile applications. In *ACM Multimedia*, 2016.

Xin Wang, Fisher Yu, Zi-Yi Dou, and Joseph E Gonzalez. SkipNet: Learning dynamic routing in convolutional networks. *arXiv preprint arXiv:1711.09485*, 2017.

Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2016.

Bichen Wu, Alvin Wan, Xiangyu Yue, Peter Jin, Sicheng Zhao, Noah Golmant, Amir Gholaminejad, Joseph Gonzalez, and Kurt Keutzer. Shift: A zero flop, zero parameter alternative to spatial convolutions. *arXiv preprint arXiv:1711.08141*, 2017.

Saining Xie, Ross Girshick, Piotr Dollr, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Conference on Computer Vision and Pattern Recogonition (CVPR)*, 2017.

Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.

Xinchuan Zeng and Tony R Martinez. Using a neural network to approximate an ensemble of classifiers. *Neural Processing Letters*, 2000.

Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. Accelerating very deep convolutional networks for classification and detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2016.

Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. ShuffleNet: An extremely efficient convolutional neural network for mobile devices. *arXiv preprint arXiv:1707.01083*, 2017.

Heliang Zheng, Jianlong Fu, Tao Mei, and Jiebo Luo. Learning multi-attention convolutional neural network for fine-grained image recognition. In *International Conference on Computer Vision (ICCV)*, 2017.

Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *International Conference on Learning Representations (ICLR)*, 2017.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2017.