VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

DEEP LEARNING FOR IMAGE RECOGNITION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MICHAL KOZEL

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

HLUBOKÉ NEURONOVÉ SÍTĚ V ROZPOZNÁVÁNÍ OBRAZU
DEEP LEARNING FOR IMAGE RECOGNITION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MICHAL KOZEL
AUTHOR

VEDOUCÍ PRÁCE Ing. MICHAL HRADIŠ,
SUPERVISOR

BRNO 2015

Abstrakt

Neuronové sítě momentálně dosahují nejlepších výsledků při rozeznávání řeči, obrazu a i dalších klasifikačních úloh. Tato práce popisuje základní prvky a vlastnosti neuronových sítí a způsob jejich učení. Cílem této práce bylo rozšířit Caffe framework o nové metody učení a porovnat jejich výsledky pomocí experimentů na datasetu Cifar-10. Konkrétně RMSPROP a normalizovaný SGD

Abstract

Neural networks are currently state-of-the-art technology for speech, image and other recognition tasks. This thesis describes basis properties of neural networks and their learning. The aim of this thesis was to extend Caffe framework with new learning methods and compare their performance on Cifar10 dataset. Namely RMSPROP and normalized SGD

Klíčová slova

neuronové sítě, hluboké učení, konvoluční neuronové sítě, rozpoznávání obrazu, Cifar-10,RMSPROP, normalizovaný SGD

Keywords

Neural networks, deep learning, convolutional neural networks, image recognition, Cifar-10, RMSPROP, normalized SGD

Citace

Michal Kozel: Deep learning for image recognition, bakalářská práce, Brno, FIT VUT v Brně, 2015

Deep learning for image recognition

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Michala Hradiše

. . . .
Michal Kozel
May 19, 2015

Poděkování

Rád bych poděkoval vedoucímu mé bakalařské práce Ing.Michalu Hradišovi ze jeho vedení, pomoc, diskuzi a trpělivost

Contents

List of Figures

Chapter 1

Introduction

Neural networks are currently widely used for various classification tasks and other AI problem. They are capable of approximating any function and becoming more powerful every day as scientists keeps developing and improving concepts and learning methods of these networks.

Over the past years top classification errors on most challenging datasets like ImageNet have significantly dropped thanks to neural networks.

Convolutional neural networks keep winning various image recognition competitions and can be considered state-of-the-art technology for such tasks.

It is said that Google has basically changed to deep learning company [12] as it uses deep learning in almost every field. From speech recognition on Android devices, image recognition for image search, to using recurrent neural networks for translation.

Another specific of this field is its openness with many deep learning frameworks to use. Almost every new discovery in this field is immediately made public and open for others to explore.

This thesis describes experiments on Cifar-10 dataset during which I was trying to enhance and to some degree optimize learning process of neural networks. To do so I have implemented two new methods of learning in caffe framework [2]. The first method is so called RMSPROP [4], originally proposed by G.Hinton as a mini-batch version of RPROP. The second method is normalized version of stochastic gradient descent.

Chapter 2 describes basic properties and components of neural networks. Following chapter 3 focuses on the learning process and what influences its performance. Chapter 4 gives an account of specifics for image recognition task. Chapter 5 introduces Caffe deep learning framework developed by BVLC[1], that I used for my experiments. Last chapters 6 and 7 describe extensions I have made to caffe framework and presents the results of my experiments with newly implemented learning methods.

Chapter 2

Neural Networks

In this chapter I will introduce basic concepts and architectures of neural networks. In the first part I will describe model of a single neuron. Then I will continue to describe how to build networks from these single neurons and various types of networks we can build.

2.1 Neurons

Neurons are the basic computational units of neural networks, originally inspired by biology (fig:2.1). Real neuron has a dendritic tree using which it receives signals from synapses of other neurons. It then sends this signal farther through its axon to next neurons. This creates a network of communicating neurons capable of solving various tasks.

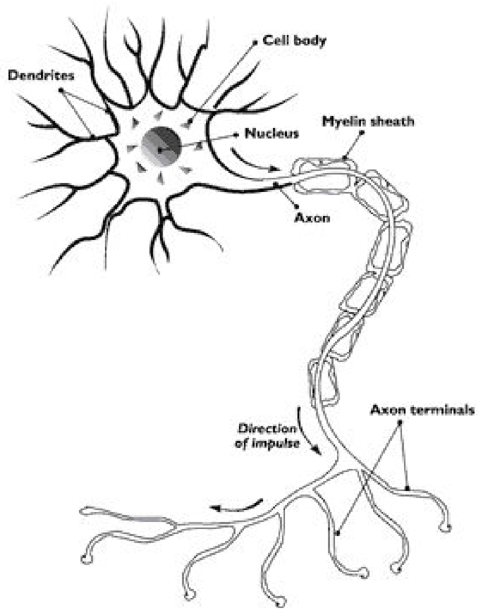


Figure 2.1: Diagram of a biological neuron. Image from:[13]

Artificial neurons works in very similar way. Basically an artificial neuron has several inputs, based on which it produces an output signal. Every input to a neuron has its weight. These weights are adjusted during learning to force the network produce desired outputs. The most simple form of a neuron is so called perceptron. A perceptron takes a number of inputs, computes their weighted sum $\sum_i w_i x_i$, and compares it with some threshold value. Based on this comparison, it then outputs either 1 or 0.

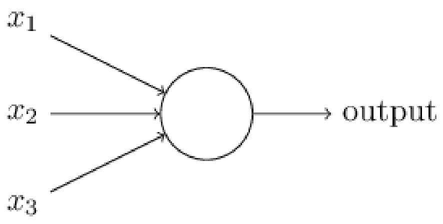


Figure 2.2: Model of a neuron. Image from:[15]

Currently used neurons are slightly different and add an activation function to this concept. Output, or so called activation, of such a neuron is then $a = f(z)$ where $z = b + \sum_i w_i x_i$. b is a bias of the neuron.

Any function can be used as an activation function. Most commonly used are sigmoid or hyperbolic tangent function. However, the rectified linear unit has been becoming very popular of late as a best neurons for image recognition [9].

Activation functions of neurons:

- Sigmoid:
$$f(z) = \frac{1}{1 - e^{-z}} \tag{2.1}$$
- Hyperbolic tangent:
$$f(z) = tanh(z) \tag{2.2}$$
- ReLU:
$$f(z) = max(0, z) \tag{2.3}$$

2.2 Architecture

By connecting single neurons we can built various neural networks. Different architectures are suitable for different tasks. The first layer is called input layer, last one is output layer and all the layers in between are referred to as hidden (fig:2.2). If the network has more than one hidden layer we call this network deep. Network can compose of one or more types of neurons.

The most common type of a neural networks is so called feed-forward network (fig:2.2). The outputs of one layer are used as the inputs of the following layer. This is also the most commonly used architecture for image recognition and in my thesis I have experimented with training of such networks.

Another type is so called recurrent neural network. This type of network includes cycles. Thanks to this feature, recurrent networks are used for modeling sequential data and time series. For example Google BRAIN project [12] very successfully uses recurrent neural networks for translators since it is mapping one sequence of words to another. This networks can also remember information for long period of time and are said to be more biologically realistic. This makes them more powerful than feed-forward networks, but also much harder to train.

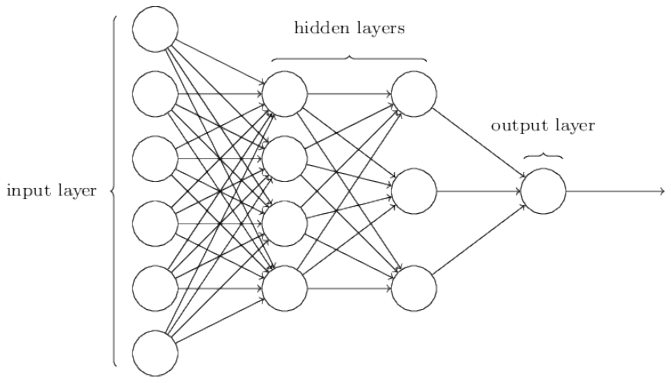


Figure 2.3: Example of simple neural network. Image from [15]

There are, of course, many other ways how to connect single neurons to form a network. In my thesis I will be focusing on feed-forward networks, mainly on their learning.

Chapter 3

Learning

This chapter describes how neural networks are actually trained. To do so I will start with description of cost function(sometimes referred to as loss or objective function). Then I'll proceed to describe general way of training networks and backpropagation, the key algorithm in this process. Lastly, I will discuss other aspects that can influence learning process.

3.1 Cost function

The cost function is basically a function of network's weights and biases that shows how well the network is performing on a certain task by computing its errors. An example of cost function is quadratic cost function:

$$f(w, b) = \frac{1}{2n} \sum_x (y(x) - a)^2 \tag{3.1}$$

w and b are weights and biases of the network. a is expected output of the network given the input x and $y(x)$ is actual output of the network given the input x. $y(x)$ also depends on current values of weights and biases. We can see from the equation that the cost function computes sum of squared differences between expected and actual outputs for a set of inputs x. When the actual outputs of the network are similar to the expected outputs, the value of cost function will be close to 0, which means the network performed well. Having said that the learning process comes down to minimizing the cost function by adjusting weights and biases of the network. This is done using gradient descent algorithm.

3.2 Gradient descent

Gradient descent is an iterative algorithm that minimizes a function by taking small steps in the direction of the steepest descent. That direction is computed by differentiating the cost function by set of weights we want to optimize. Gradient vector of two-dimensional function shown on picture 3.2 can be therefore computed as follows:

$$\nabla C = (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T \tag{3.2}$$

The gradient measures the impact of the variable change on the cost function. The desired change of vector $v = (v1, v2)$ is then:

$$\delta v = -n\nabla C \tag{3.3}$$

n is called learning rate and determines the size of step. The learning rate is significant parameter of gradient descent algorithm. As you can see on picture 3.2 magnitudes of gradient will vary a lot depending on the position. This means gradient descent tends to do bigger steps on positions where gradients are big and will do only little changes on places with tiny gradients. With big learning rates, the algorithm can keep overshooting the minimum of function or even diverge. For small learning rates the learning will be very slow. I have tried to address this issue later in this work at chapter 6.

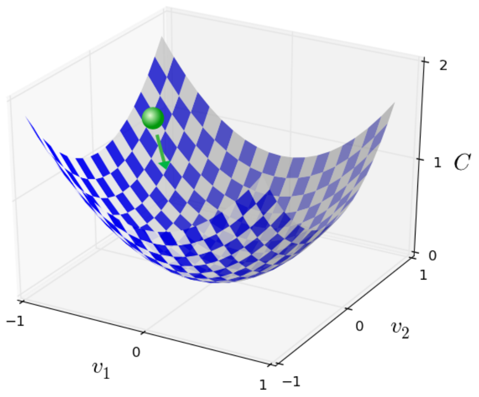


Figure 3.1: Illustration of gradient descent, Image from [15]

Another question is how often to update weights. This divides learning into three categories [7]:

Online
Weights are updated after each training case.

Full-batch
Weights are updated after a full sweep through training data.

Mini-batch
Weights are updated after a small(random) sample of training cases.

We can choose to end gradient descent learning when the network shows signs of overfitting 3.5, after fixed number of iterations or when the accuracy is no longer improving.

3.3 Backpropagation

Methods like gradient descent use gradients of cost function in order to minimize its value. These gradients measures how a change in particular activation influences the cost function. Backpropagation is an effective algorithm to compute such gradients. The change in one activation also affect many other activations in the network [7]. These effects must be combined. Backpropagation is very similar to the forward pass in a network. Forward pass takes an input, keeps computing activations of single neurons layer by layer until it gets to the last layer and computes output. What backpropagation does is that it computes these gradients of weights for the last layer and propagates them back through the network using same weights as forward pass.

Particularly for deep networks, the vanishing gradient problem can occur. Derivatives of activation functions of single neurons are used during this process. For example for sigmoid, neurons these derivatives are close to 0 for either too big or too small values. Gradients, therefore, can be vanishing during backpropagation, causing little to no updates of weights in lower layers. Similarly an exploding gradients problem could occur. This can be addressed by choosing suitable activation function, or also by normalizing update values as I have done in chapter 6. Backpropagation algorithm is defined as follows [15].

First we need to perform forward pass to compute activations of single layers. Then we compute error of the output layer:

$$\delta^L = \frac{\partial C}{\partial a^L} f'(z^L) \tag{3.4}$$

L is index of last layer. f is activation function of neurons and z, as before, is sum of weighted inputs to such neurons. The next equation computes error of a layer using the error of following layer:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot f'(z^l) \tag{3.5}$$

$\odot$ is Hadamard product [14]. It takes two matrices A, B of same sizes and computes matrix C where for each element of C we can say $C_{ij} = A_{ij}B_{ij}$ The last two equations computes gradients with respect to particular weights and biases:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{3.6}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{3.7}$$

This way we only need one forward pass and one backward pass to compute all gradients. Others methods are based on changing random weight, run forward pass to see the influence of that weight on cost function, and keep that weight updated if the influence was positive. Such method are very inefficient compared to backpropagation.

3.4 Weight initialization

We have to choose the initial set of weights before we start training. There are many approaches how to do so [6]. The key thing is to start with different weights. Neurons with same incoming and outgoing weights would get same gradients during the learning. As describe in the regularization section (3.6) it is better to keep weights as small as possible. The weights are usually initialized to some small numbers close to zero. One possibility is to initialize weights with random Gaussian variables with standard deviation 1 and mean 0. Another way, recommended by Hinton [7], is to choose weights as a proportion to square root of unit fan-in. Weight initialization can have huge effect on learning process [15].

3.5 Overfitting

With so many parameters neural networks can fit basically any function. This can lead to phenomenon called ovefitting. As described before the aim of learning is to minimize cost function. However, all this is done using training data. If we look at a cost function on testing data, it can, at some point, stop to decrease and start to increase. Another sign of overfitting is raising accuracy on training data but stagnating accuracy on testing data. This all means our model fits too well on training data and will be unable to generalize. At that point the network can be learning some accidental regularities in our data (If we provide only pictures of red cars, the network may infer that all cars are red and classify cars of other colors incorrectly. This is a typical consequence of overfitting).

One way to reduce overfitting is to simply use networks with smaller capacity. Although it can reduce overfitting, smaller networks are generally less powerful and unsuitable for some tasks.

By far the best way to deal with overfitting is to get more training data. This is not always possible. We can at least artificially expand training data by tilting images, or flipping them around their vertical axis. This method proved to be very useful. There are also some other methods that can help to reduce overfitting. I will describe them in the following sections.

3.6 Regularization

The most commonly used regularization method is weight decay or so called L2 regularization. The idea behind this is adding an extra term to the cost function. This term is sum of the squares of all the weights in the network multiplied by $\lambda/2n$ where λ is a regularization parameter and n is size of training set .

$$\frac{\lambda}{2n}\sum w^2 \tag{3.8}$$

The gradient descent algorithm will now be trying to minimize this amended cost function. The added term basically penalizes huge weights. By increasing λ we would be adding importance to this added term, therefore making the gradient descent focus more on keeping small weights. The theory is that by keeping small weights the network basically learns simpler hypotheses and therefore generalize better.

Almost the same approach is L1 regularization. The added term to cost function is following:

$$\frac{\lambda}{n}\sum |w| \tag{3.9}$$

The difference is that in L1 regularization the weights are reduced by a constant amount whereas in L2 regularization they are reduced proportionally to their magnitudes [15].

3.7 Momentum

Momentum is another method used to enhance learning process. Good analogy of momentum is a velocity of a physical object. If we apply a force to such object, it will not just move in the direction of this force but the force will be combined with the object's current velocity. Using same principle we will not update weights by just current gradient but we will combine it with its previous update. So at first we will compute current velocity:

$$v_t = \mu v_{t-1} - n\nabla C \tag{3.10}$$

and the update the weights using this velocity:

$$w_t = w_{t-1} + v_t \tag{3.11}$$

Previous velocity is multiplied by μ, which can be thought of as a friction. μ is usually set to small value on the beginning of training because gradients here tend to fluctuate a lot [7]. From this initial value, usually around 0.5, we increase μ during the training typically to 0.99. This method reduces fluctuations and increases the length of steps in directions in which the gradients are stable. As as result the learning is much faster and stable, which allows us to use bigger learning rates. This properties makes momentum almost a necessity. Combination of momentum and SGD is therefore one of the most commons ways to train neural networks [7].

Chapter 4

Neural networks for image recognition

This chapter describes methods that are widely used for image recognition tasks. One of the main problems concerning this tasks is the position of object on the picture. Convolutional networks are used to deal with this problem. Another methods that proved to be beneficial for image recognition include dropout or using of rectified linear units.

4.1 Convolutional neural networks

Convolutional networks are feed-forward networks that have so called convolutional layers typically followed by pooling layers. This type of networks achieve the best results in image recognition tasks[9]. As you could see at figure 2.2, the output of a neuron in a feed-forward network is used as input of every neuron in the following layer. Convolutional networks use sparse connectivity as shown on figure 4.1. Neuron with such restricted connectivity is then called filter. To ensure viewpoint invariance such filters are replicated to cover entire visual field to be able to detect features regardless of their position. Those filters share same weights and biases and forms so called feature maps (fig:4.1). The sharing also significantly reduces number of free parameters in the network.

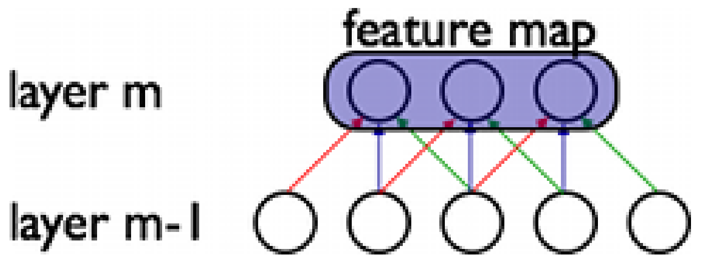


Figure 4.1: Feature map in convolutional network

Each convolutional layer learns to detect different features. The very first layer usually learns to detect edges, the following layers can detect more complex shapes or objects (fig:4.1).

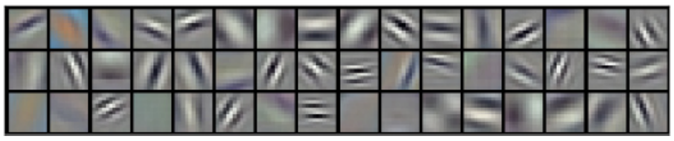


Figure 4.2: Example filters learned to detect edges

4.2 Pooling layer

There are two basic types of pooling layers: max and average. Pooling layers, depending on their type, compute max or average value of their input (for example a region of an image). This reduces the resolution of the input and provides some spatial invariance. However, as a consequence we lose information about precise position of the features. This is not a problem for some object recognition task but such information can be crucial for, for example, face recognition, where we need to know exact proportions of face to correctly identify the person on the image. Pooling regions can also overlap.

4.3 Dropout

This method was introduced by G.Hinton and is another way of preventing overfitting [8]. The idea behind it is simple. During the learning process neurons from hidden layers are omitted with some probability, typically 0.5. These neurons then do not contribute to neither forward pass nor backpropagation. This forces neurons not to depend too much on some inputs as they can be missing. This significantly improves generalization of such network. It can be seen as an equivalent to training 2^n networks, where n is the number of hidden units.

The similar approach can be used for input layer. In this case we can omit say 20% of pixels from input image. As I mentioned before (3.5) overfitting can be also reduced by using networks with smaller capacity. However G.Hinton suggests to rather use networks with bigger capacity and use dropout [7].

I was not using this method during my experiments described in chapter 7. From my previous experience, it improves accuracy of similar networks on Cifar-10 dataset.

4.4 Rectified Linear Units

I have briefly mentioned rectified liner units in section 2.1. The activation function is $f(x) = max(0, x)$. ReLU turned out to be the best choice for recognition tasks [9]. The advantages are that networks with ReLUs can be trained several times faster and can get sparse representations of inputs.

4.5 Softmax

Softamx is the very last layer of networks used for classification. It assigns probabilities to all outputs of such network. For example how probable it is that the input image falls into class car, airplane, bike ... The sum of all probabilities have to be 1 and the classes needs to be mutually exclusive. Softmax layer computes its outputs as follows:

$$y_i = \frac{e^{z_i}}{\sum\limits_k e^{z_k}} \tag{4.1}$$

It simply computes proportion of each output by dividing it by sum of all outputs.

Chapter 5

Caffe framework

This chapter describes Caffe framework which I used for my experiments. Caffe [2] [11] is a deep learning framework developed by Berkeley Vision and Learning Center (BVLC) [1]. The project stared as a PhD thesis of Yangqing Jia [10]. Caffe enables to train various neural networks on a GPU and can process over 60M images per day with a single NVIDIA K40 GPU. It also have an extensive community of contributors and comes with many examples and tutorials. Caffe also provides scripts to download the most popular datasets (MNIST, CIFAR-10, and ImageNet) and to convert them to lmdb databases for which Caffe is optimized.

Caffe is a fast C++/CUDA implementation of a deep learning framework and uses BLAS as a backend for its vector and matrix computations. However, optional Mathlab or Python interface can be installed.

5.1 Caffe interface

Two configuration files are used to specify everything needed for training of a neural network.

The first file defines the network, layer by layer. Caffe provides implementations of most common neurons(sigmoid, hyperbolic tangent, and ReLU) combined with various layers including convolution and pooling layers, dropout, fully connected layer or softmax. Additional parameters like pooling type (MAX,AVG), filter size for convolution layers, or number of outputs from layer are also set here. Data layer defines batch size and path to the data set on which we want to train our network.

The second file, so called solver, specifies global parameters for learning. For instance learning rate, learning rate policy, momentum, number of iterations or testing interval. Learning method is also set here. Basic learning method in Caffe are SGD(stochastic gradient descent), NESTEROV(Nesterov's Accelerated Gradient) and ADAGRAD(Adaptive Gradient). Additionally I have implemented NSGD(Normalized SGD) (6.1) and RMSPROP (6.2).

5.2 Training a network

If everything was set up correctly Caffe initializes specified network and begins training. The training is carried out as described earlier (3) using backpropagation to compute gradients and then update the weights according to the chosen method. During the process Caffe prints information about current iteration, value of cost function or accuracy on the testing data. For cases when something goes wrong, Caffe can create a snapshot to store all necessary information about training. The snapshot can then be used to resume training. We can also choose to resume training using modified solver file, for example with changed learning rate or learning method.

Chapter 6

Extensions to Caffe

As a part of my experiments I have implemented two learning methods for the Caffe framework. The main aim was to normalize weight updates in the gradient descent learning process. The reason for this is that the magnitude of gradients can vary between different weights and can change often and unpredictably during the learning. This can lead to either too big or too small weight updates and it also makes it very hard to choose a learning rate that would be suitable during whole learning process. Normalizing should mitigate this issue. It can also make learning faster and more stable. I found two ways how to do so.

The first idea is to use only the signs of gradients and update every weight by a fixed value. This idea is the base for method called rprop, which was later enhanced by G.Hinton to rmsprop [4].

The second approach is to divide the vector of gradients by the Euclidean norm of this vector. This ensures that the length of this vector becomes 1. By updating the weights by such a vector we take a step of constant size in the direction of the steepest descent on the error surface.

6.1 Normalized Stochastic Gradient Descent (NSGD)

As implied above, the goal is to be taking steps of a constant size on the error surface. This should eliminate huge updates that appear when the gradients are big, which can cause divergence as described earlier (3.2). This method should also considerably speed up learning on plateaus with tiny gradients.

During the backpropagation Caffe computes a vector of updates for every layer. The Euclidean norms of these vectors are computed using BLAS library as follows:

$$|x| = \sqrt{x_1^2 + x_2^2 + ... + x_n^2} \tag{6.1}$$

By dividing those vectors by their Euclidean norms we obtain the normalized vectors of length 1. These vectors are, as in usual SGD, multiplied by the learning rate and then used for weight update. This means the size of every step is given by the learning rate and is same during whole learning process.

Another possibility is to divide the update vectors by the square roots of their Euclidean norms. It is simple to prove that in this case the lengths of resulting vectors will not be 1 but rather the square root of their original lengths. This still speeds up learning on the plateaus, reduces the step size for huge gradients, but also takes into account the magnitudes of gradients, which proved to be beneficial in some cases. The detailed results are described in the experiments chapter 7. This version will be from now on referred to as NSGD2.

6.1.1 Adapting step sizes

As another improvement I tried to assign separate learning rates for every layer and keep updating those rates throughout the learning process. The idea was to keep track of changes in every layer and adjust the learning rates to improve the learning. For instance if the changes that where needed for a particular layer were repeatedly huge, I would increase the learning rate for this layer. On the other hand, if almost no change was needed I would decrease the learning rate for this layer. This seems reasonable for example for CNNs, where the first layer can learn edge detection and we do not want to change the weights here anymore. However, this introduces many parameters to tune (additive/multiplicative decreases/increases of learning rates, conditions for when to update learning rates, and others). Although it can be tuned to improve learning, it did not proved to be beneficial in the majority of cases. For these reasons I have left the implementation commented in the code, but I did not use it for my experiments as it requires further investigation.

6.2 RMSPROP

This method is based on rprop, which combines the idea of only using the sign of the gradient with the idea of adapting the step size separately for each weight. This idea came along for the same reasons I described earlier: stability and speed of learning. The problem here is that the simple SGD averages the gradients over successive mini-batches, which of course is not the case of rprop [7]. If, for instance, the gradient of nine mini-batches is roughly +0.1 and gradient of next mini-batch is -0.9, rprop would increment the weight nine times by the same value and decrement it just once, again by the same value. Unfortunately, it is not what we want as in this case the weight should stay the same. This is the reason why rprop cannot be used for mini-batch learning, only for full-batch.

This problem was solved by introducing RMSPROP by G.Hinton [4]. Rmsprop keeps the moving average of squared gradients:

$$MeanSquare(w,t) = 0.9 * Meansquare(w,t-1) + 0.1 * (\frac{\partial C}{\partial w}(t))^2 \tag{6.2}$$

The vectors of gradients, that we use to update the weights, are divided by the square root of this moving average before the update. This forces the numbers we divide by to be very similar for adjacent mini-batches and consequently making learning possible. Similar approaches were also suggested by LeCunn [16].

6.3 Implementation

I have implemented both these methods in Caffe framework and performed various experiments with them. The implementation of NSGD has both its variants and both of them were also used for the experiments.

Chapter 7

Experiments with SGD, Normalized SGD and RMSPROP

To try my implementations of described methods I was experimenting on the Cifar-10 dataset. Results of these methods can widely vary for different network architectures, hyper-parameters and lots of other factors. My goal here was not to achieve best classification results or to train the deepest network but rather compare these methods. As I mentioned before the result may vary and different methods can work better for different architectures. For that reason I chose one of the most common architectures used for Image recognition, described at 7.1 and performed my all experiments on it.

I trained the same network using SGD, normalized SGD, and RMSPROP with learning rates 0.01, 0.001 and 0.0001. These learning rates were experimentally selected as suitable for this task. I chose to focus on learning rates because they correspond to the step size for normalized learning. This way I could also compare the speed and stability of these learning methods as they heavily depend specifically on learning rate. I performed every training over 80 000 iterations, which corresponds to 160 epochs. An epoch is finished once all of the training pictures were used. Every training was performed tree times and I present the best achieved results. Achieved accuracies are presented in tables in two columns: Accuracy on the end of learning and top accuracy during the learning. This is due to the fact that overfitting (3.5) may have occurred because of fixed number of learning epochs for various learning rates.

7.1 Architecture

Picture 7.1 shows the typical architecture widely used for image recognition tasks. The first layers are convolutional and are expected to learn to detect particular features. The very first layer typically learns to detect edges. The following convolutional layer will learn to detect more complex features like shapes. These layers are interleaved by pooling/sub-sampling layers. Typically one or more fully connected layers follow. The very last layer depends on the task the net was designed for. For image recognition it's usually softmax which assigns probabilities to possible results.

Concrete architecture I have used for my experiments is very similar to the one described above. First layer is convolution followed by max pooling layer. This is followed by another two convolution layers but this time combined with average pooling layers. Experimentally I chose to use two fully connected layers. The very last layer was softmax.

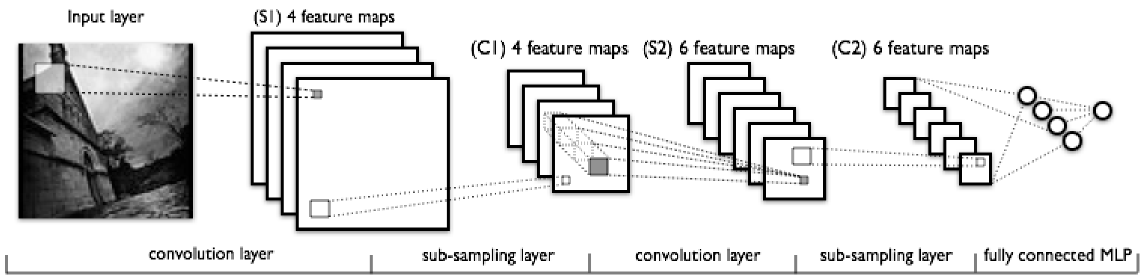


Figure 7.1: Typical architecture of CNNs where convolution layer are followed by pooling layers. Image from [5]

7.2 Cifar10

Cifar-10 [3] contains 60 000 color images. The whole dataset is divided into 50 000 training images and 10 000 testing images. Every image has a size of 32 x 32 pixels and falls into one of 10 equally big categories. The list of categories with sample images is on picture 7.2. Top achieved accuracy on this data set is 85% and was achieved using Bayesian hyper-parameter optimization[3].

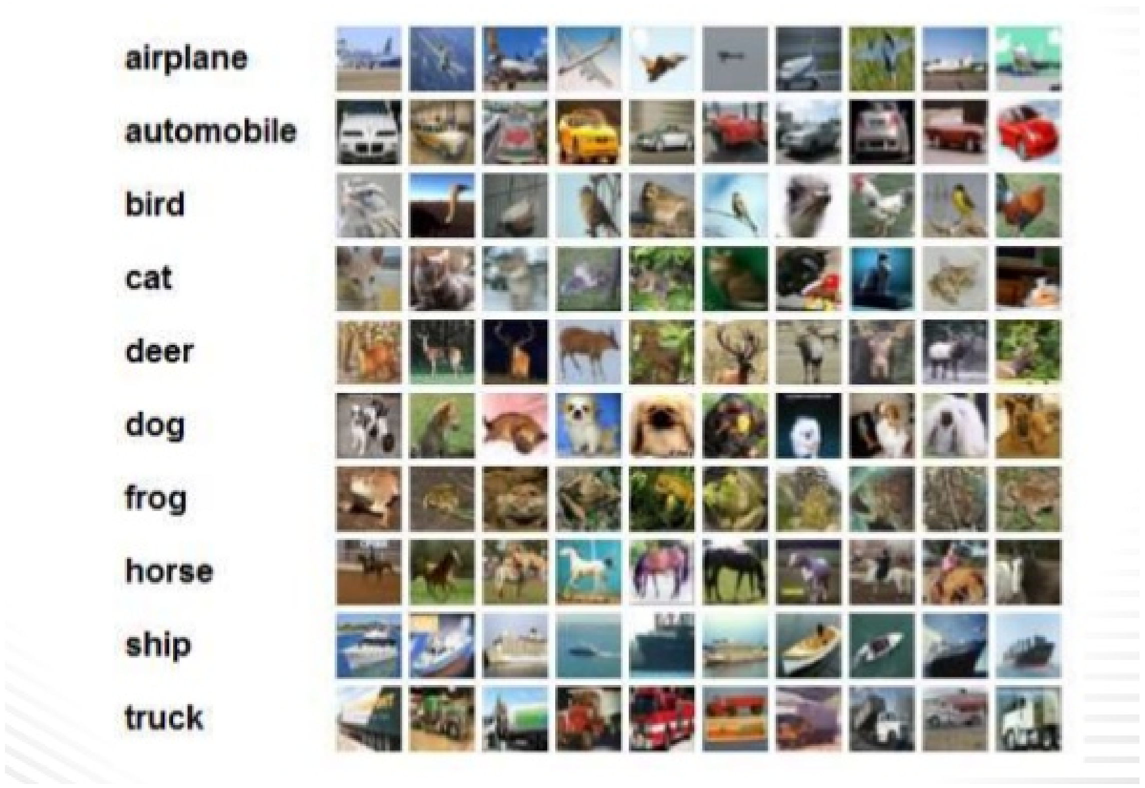


Figure 7.2: Cifar-10 categories with sample images

7.3 SGD

SGD is the most commonly used method for training neural networks. It is the mini-batch version of Gradient descent. In the experiments I used SGD combined with momentum and L2 regularization. The parameter of momentum was set to 0.9. The weight decay for regularization was set to 0.004. I have achieved following results:

learning rate	accuracy on the end	best achieved accuracy
0.0001	74.4%	76.0%
0.001	74.5%	74.8%
0.01	10%	10%

Table 7.1: Table showing accuracies of SGD trained networks

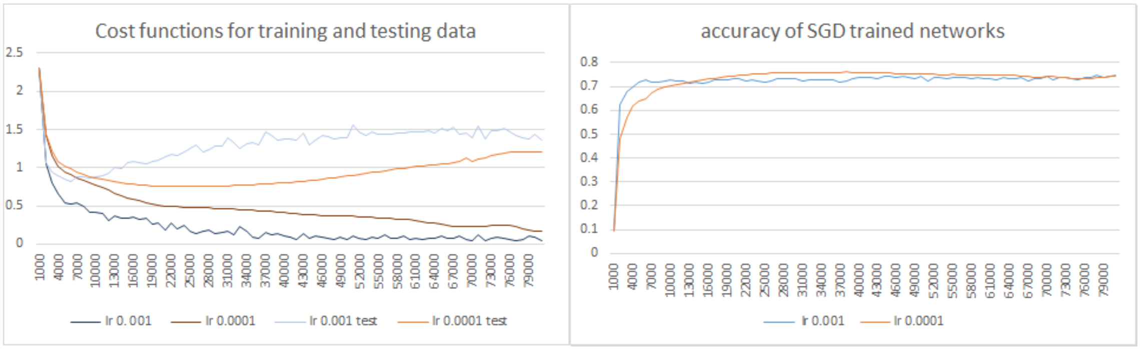


Figure 7.3: Process of learning using SGD

The learning went as expected. It started of fastest for the 0.001 learning rate but on the end fall a bit short behind the top accuracy of 0.0001. 0.01 proved to be too big of a learning rate and failed to train the network. Values of cost functions in figure 7.3 shows that overfitting occurred. Training with even lower learning rate (0.00001) was too slow to get the accuracy over 71% inside 160 epochs. Without considering adapting learning rate during the training, we can conclude that the best learning rate for this particular setting is 0.0001.

7.4 Normalized SGD

Normalized SGD is one of the methods I have added to Caffe framework and described here: 6.1. Both described variants were used for experiments. Similarly to SGD training, I have also combined this method with L2 regularization with 0.004 weight decay parameter and a momentum with 0.9 parameter. I have achieved following results:

Figure 7.4 shows the results of basic version of Normalized SGD. Training with the 0.0001 learning rate slowed down considerably compared to the basic version of SGD but showed solid results after 160 epochs of training. Training with the 0.001 learning rate went very similarly to basic SGD with 0.0001 learning rate and achieved almost the same top accuracy. This method was able to train the network even with the 0.01 learning rate. In that case the network did not reach as high accuracies as SGD. The best learning rate for this particular setting is 0.001. Overfitting for some rates is also visible on the graph

learning rate	accuracy on the end	best achieved accuracy
0.0001	71.7%	71.7%
0.001	73.5%	76.2%
0.01	71.3%	73.3%

Table 7.2: Table showing accuracies of NSGD trained networks

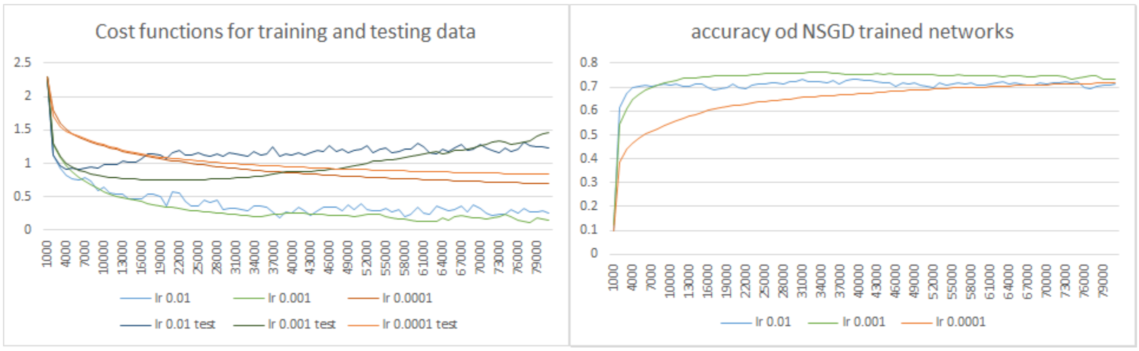


Figure 7.4: Process of learning using NSGD

Figure 7.4 shows the results of the second version of Normalized SGD. The results here were very similar to the first version of this method but quite a bit faster. Comparing the result of the first and the second version of Normalized SGD with the 0.001 learning rate, the second version reached classification accuracy of 70% after 4000 iterations, whole 4000 iteration earlier than the first version. The massive speed up is also apparent for the 0.0001 learning rate. This particular combination was the only one that was both fast and stable enough to get the classification accuracy over 77% inside 160 epochs of learning. As apparent from the graph the accuracy for this learning rate would probably get even higher after longer learning. This time overfitting appeared only for the highest learning rate.

learning rate	accuracy on the end	best achieved accuracy
0.0001	77.2%	77.2%
0.001	74.2%	74.7%
0.01	70.9%	72.2%

Table 7.3: Table showing accuracies of NSGD2 trained networks

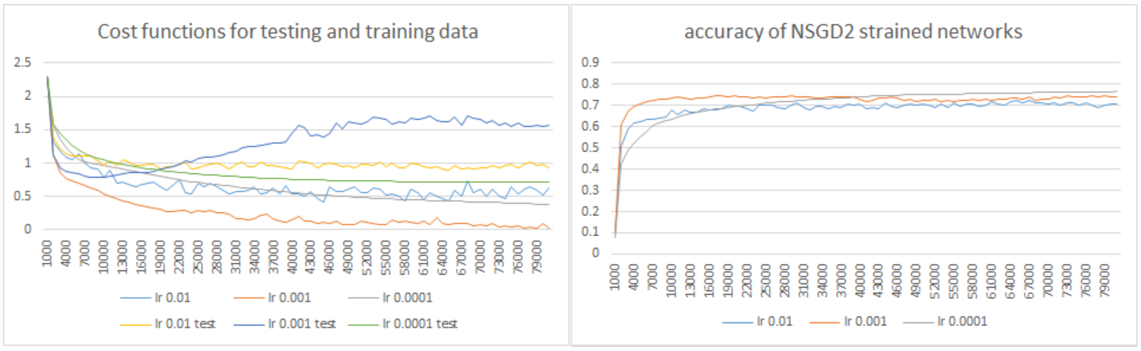


Figure 7.5: Process of learning using NSGD2

7.5 RMSPROP

RMSPROP as proposed by G.Hinton [4]. The L2 regularizaton was also used here with 0.004 parameter. The momentum was not used as it doesn't have that good infuence on RMSPROP [4]. Although very different, this method showed very comparable results with the others. The learning for 0.0001 and 0.001 learning rates stared off with the same pace. However, after 7000 iterations the learning process with the lower rate continued to improve network's accuracy and reached slightly above 75%, leaving the learning with the higher learning rate behind. Similarly to SGD, learning rate 0.01 proved to be too big for this method. Accuracies of RMSPROP stayed behind all other tested methods. The better choice of learning rate in this case was 0.0001.

learning rate	accuracy on the end	best achieved accuracy
0.0001	72.5%	75.5%
0.001	68.5%	70.4%
0.01	10%	10%

Table 7.4: Table showing accuracies of RMSPROP trained networks

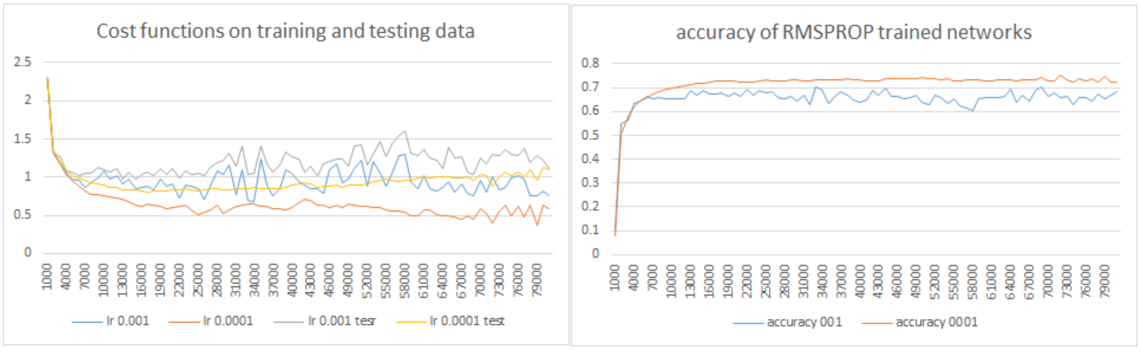


Figure 7.6: Process of learning using RMSPROP

7.6 Comparison and conclusions

All tested methods achieved comparable results on Cifar-10 dataset. The outcome here is that any method, as long as we chose the correct hype-parameters, can achieve solid results. This is demonstrated by the graph 7.6 that compares the best learning curves for every method. Further tuning of hyper-parameters could probably increase the accuracies a bit. The Normalized SGD was able to perform training with bigger learning rates than ordinary SGD and RMSPROP but suffered from slow down for lower learning rates. The second version of normalized SGD(NSGD2) turned out to be a sensible compromise between SGD and NSGD. It is also the method that achieved the best classification accuracy inside the 160 epoch of learning. This method also showed potential to increase the accuracy even more in longer learning. NSGD2 can be considered a success as it has outperformed even most commonly used methods.

method	best achieved accuracy
SGD	76.0%
NSGD	76.2%
NSGD2	77.2%
RMSPROP	75.5%

Table 7.5: Comparison of top accuracies achieved by different methods

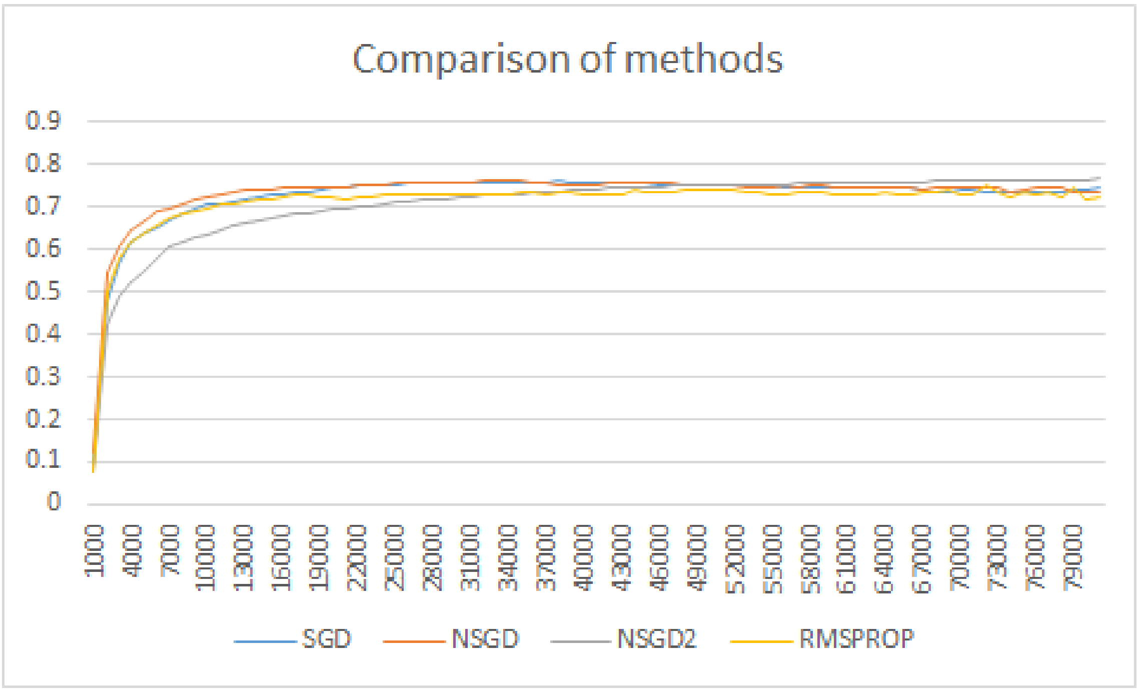


Figure 7.7: Comparison of different learning methods. Graph shows achieved accuracies

Chapter 8

Conclusions and future work

This thesis described basic concepts of neural networks and focused mainly on learning methods for deep neural networks. I have extended Caffe framework by two learning methods: normalized SGD and RMSPROP. Subsequently I compared their performances. Both of them proved to be comparable or, in some cases, even outperformed the most commonly used method: SGD. Particularly NSGD2 showed potential and could be explored further. The methods should be also tested using different datasets and network architectures.

Dynamic adjustments of learning rates during the training (6.1.1) still needs more investigation. Although it could be tuned to improve learning process, in the majority of cases it made learning impossible. In such state it would only introduce more hyper-parameters to tune without any really significant improvement to whole process. By this approach I was trying to replace technique, where we, either continuously or in steps, decrement the learning rate during the learning process. This technique is usually able to reach better results, inside a certain number of training epochs, than the fixed learning rate. I have excluded this technique from my experiments as I would make the comparison of learning methods harder and rather confusing.

Another possibility is to try normalizing step size for another learning methods like Nesterov accelerated gradient. This would probably have similar effect as normalizing had on basic SGD. But NSGD2 with Nesterov seems to be promising combination because Nesterov proved to be superior to SGD in some cases. This, however, needs to be tested.

Momentum brings significant speed improvement to learning so it would be beneficial to investigate and optimize momentum separately for each particular method. In my experiments momentum was influenced by the normalizing effect because the velocity it remembered from previous updates was the normalized step of NSGD.

Initial weight initialization or even network pretraining both play important roles in learning process. I have yet to investigate influence of these on the learning methods I have implemented. I assume that it could significantly improve them as any other methods.

Taking all these things into account leaves us with huge variety of combinations to explore.

Hyper-parameters still remain a nettlesome problem as there is no universal approach how to set them correctly. We usually just have to experiment to find a suitable values, without any guarantee that they are optimal. However some algorithms can help us with setting of them. The importance of the Hyper-parameters can be seen on Cifar-10 where a 3% accuracy improvement was achieved just by using Bayesian hyper-parameter optimization to find better setting for hyper-parameters [3].

There is still lot of research to be done on this topic. The goal is to find one learning method that would be able to reliably train any network without relying too much on correct setting of many hyper-parameters. This appears to be very difficult due to wide variety of network types, including recurrent neural networks, or various network architectures. Normalizing can be a step in the right direction.

Bibliography

[1] Berkeley vision and learning center. http://bvlc.eecs.berkeley.edu/.

[2] Caffe framework. http://caffe.berkeleyvision.org/.

[3] The cifar-10 dataset. http://www.cs.toronto.edu/~kriz/cifar.html .

[4] lecture 6. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf/.

[5] Lisa lab. convolutional neural networks (lenet). http://deeplearning.net/tutorial/lenet.html.

[6] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures, version 2, sept. 16th, 2012. http://arxiv.org/pdf/1206.5533v2.pdf.

[7] Hinton Geoffrey. Neural networks for machine learning [online]. https://www.coursera.org/course/neuralnets/.

[8] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. http://arxiv.org/pdf/1207.0580.pdf.

[9] Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? http://yann.lecun.com/exdb/publis/pdf/jarrett-iccv-09.pdf.

[10] Yangqing Jia. Author of caffe. http://daggerfs.com/.

[11] 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*, 2014.

[12] Robert McMillan. Inside the artificial brain that's remaking the google empire. http://www.wired.com/2014/07/google_brain/.

[13] Elizabeth Million. Edible neuron diagram. http://www.education.com/science-fair/article/edible-neuron-diagram/.

[14] Elizabeth Million. The hadamard product. http://buzzard.ups.edu/courses/2007spring/projects/million-paper.pdf/.

[15] Michael Nielsen. Neural networks and deep learning. http://neuralnetworksanddeeplearning.com/.

[16] Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. http://yann.lecun.com/exdb/publis/pdf/schaul-icml-13.pdf.

Appendix A

Content of CD

- This thesis in PDF format and as LaTeXsource code
- Caffe framework extended by NSGD2 and RMSPROP
- Ready Cifar-10 and MNIST datasets
- Sample solver and network definition files
- The poster in PNG format

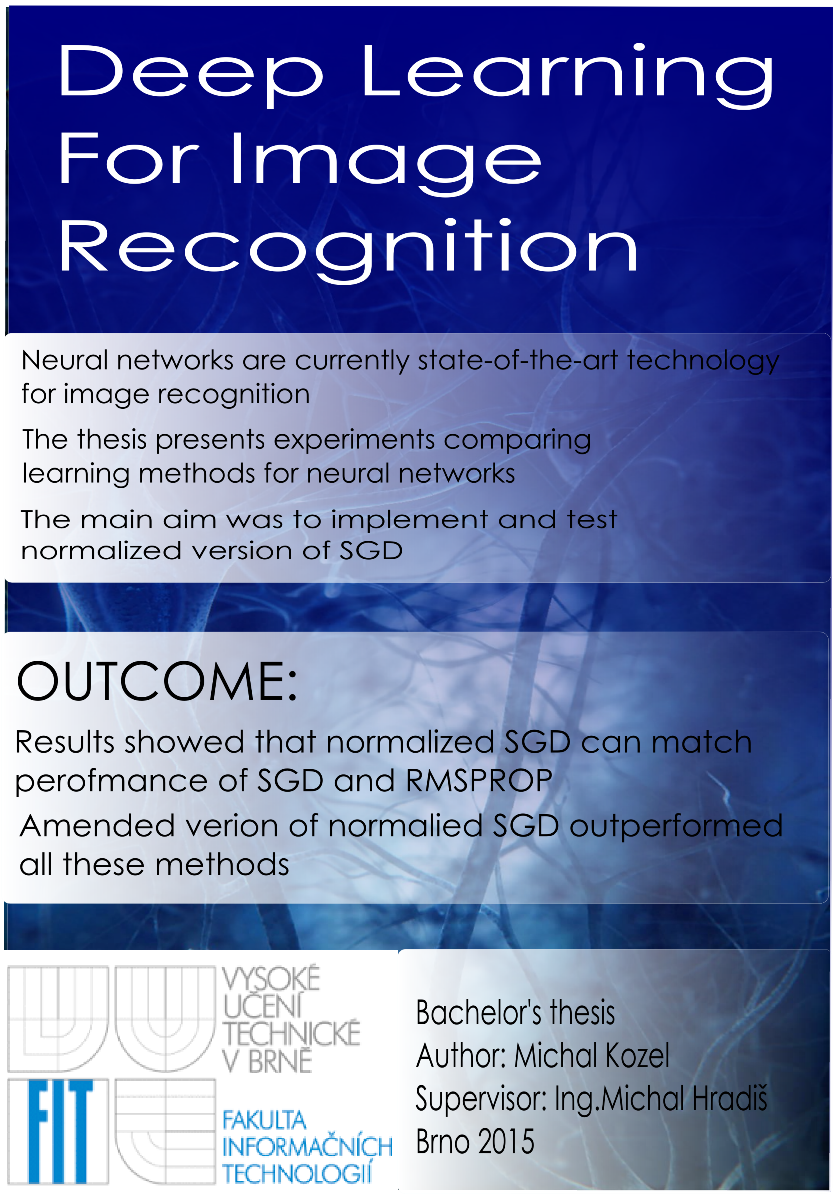
Appendix B

Manual

- Install prerequisites for Caffe framework described at installation section on Caffe website[2]
- Build Caffe from source files on CD
- Ready Cifar-10 and MNIST datasets
- Train sample network by running ./examples/cifar10/train.sh from caffe root folder
- Change file ./examples/cifar10/cifar10_bc_solver.prototxt (change solver_type to NSGD or RMSPROP to try new methods)

Appendix C

Poster



Deep Learning For Image Recognition

Neural networks are currently state-of-the-art technology for image recognition

The thesis presents experiments comparing learning methods for neural networks

The main aim was to implement and test normalized version of SGD

OUTCOME:

Results showed that normalized SGD can match perofmance of SGD and RMSPROP

Amended verion of normalied SGD outperformed all these methods

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FIT FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Bachelor's thesis
Author: Michal Kozel
Supervisor: Ing.Michal Hradiš
Brno 2015