DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Training Deep Convolutional Neural Networks on the GPU Using a Second-Order Optimizer

Mihai Zorca

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Training Deep Convolutional Neural Networks on the GPU Using a Second-Order Optimizer

Training von Deep Convolutional Neural Networks auf der GPU mit einem Optimierer Zweiter Ordnung

Author:	Mihai Zorca
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Severin Reiz, M.Sc.
Submission Date:	July 15th, 2020

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

July 15th, 2020 Mihai Zorca

Acknowledgments

I am grateful to my advisor Severin Reiz for his valuable advice and constant support over the course of this thesis.

I would like to thank my parents for raising me and providing great role-models, and for their ongoing love and support.

"The effort of using machines to mimic the human mind has always struck me as rather silly. I would rather use them to mimic something better."

-Edsger W. Dijkstra

Abstract

Deep Convolutional Neural Networks (CNNs) are a prominent class of powerful and flexible machine learning models. Training such networks requires vast compute resources: due to the large amount of training data and due to the many training iterations. To speed up learning, many specialized algorithms have been developed. First-order methods (using just the gradient) are the most popular, but second-order algorithms (using Hessian information) are gaining importance. In this thesis we give an overview over the most common first-order optimizers and how they are used to train networks. Then we build upon a sample second-order algorithm which we call EHNewton (Efficient Hessian Newton). We have integrated this into the TensorFlow platform, such that the new method can act as a drop-in replacement to standard algorithms. We make use of this, by training one CNN model each out of the (1) Inception, (2) ResNet and (3) MobileNet architectures. Due to technical limitations we limit this study to last layer training on a single NVIDIA Titan GPU.

EHNewton shows speed-up and accuracy benefits compared to first-order training algorithms on all three CNNs using the ImageNet database.

Contents

1 Introduction

Neural Networks are powerful and highly flexible models. Given enough hidden units, they can approximate any real-valued function [1]. *Convolutional* Neural Networks (CNNs) are specialized models, well suited for image recognition and other visual tasks. CNN models have even achieved superhuman performance on certain recognition tasks [1].

One of the main challenges in deep learning is training such models. The resource requirements are very large: investing multiple days to train a single network is not uncommon [1]. Over the last decade, specialized algorithms have been developed to solve this problem. The most popular methods are first-order optimizers, like SGD, AdaGrad, RMSProp or Adam, that use only gradient information. But recently, powerful second-order optimizers (also using Hessian information) have been presented [2]. They have not displaced first-order algorithms and we suspect an important reason for this is their increased complexity.

In a recent master's thesis [3], a comparatively simple second-order method has been proposed. Here, we clarify the differences between that approach and similar existing algorithms and call the new method EHNewton. Then we provide a practical implementation of the algorithm in TensorFlow and compare its training performance to first-order optimizers.

In Chapter 2, we first give an overview of Neural Networks and the challenges in training them. Then we present the most commonly used first-order methods before focusing on second-order optimization and the sample EHNewton method. Finally, we give a theoretical background on the main building blocks of three modern Convolutional Neural Network architectures.

Chapter 3 presents the TensorFlow (TF) platform and its computational graph model. Then it presents the two important high-level libraries Keras and TF-Slim, before finally explaining how we integrated EHNewton into TF.

In Chapter 4 we explain the training setup and the limitations of our EHNewton implementation. Then we train the last layer of InceptionV3, ResNet-50 and MobileNetV2 on an ImageNet dataset. Finally, we compare their performances and conclude the effectiveness of EHNewton on training deep Convolutional Neural Networks.

2 Theoretical Background

This chapter gives an overview of neural networks and the challenges in training them. Then it presents the optimization algorithms most commonly used to try to overcome these challenges, along with the new second-order algorithm. These algorithms can be used to train many kinds of neural networks, we focus on convolutional neural networks. Three state-of-the art CNN architectures of models trained in this thesis are shown in the final section of this chapter.

Note : This chapter touches on theory of neural networks and their optimization algorithms. In the Seminar Paper ”Numerical Optimization for Neural Networks”, handed in internally at TUM in 2019, we already included theory sections on many of these topics. The paper had sections on neural network training, its challenges and the commonly used algorithms SGD, AdaGrad, RMSProp and Adam. Therefore, subsections 2.1.1, 2.1.3, 2.1.4, 2.2.1, 2.2.2, 2.2.3, 2.2.4 and 2.3.1 will inevitably show some similarities in content and structure to their Seminar-Paper counterparts.

2.1 An Overview of Neural Networks

Neural Networks (NNs) are the ”quintessential deep learning models” [1]. In this section we present their structure, introduce the special convolutional layer and give an overview over the steps and challenges in training neural networks.

2.1.1 Notation

Neural networks, or multilayer-perceptrons, consist of connected layers of processor units, called *neurons* [4]. This idea of layers-of-neurons is inspired from neuroscience [1]. We call a Neural Network *deep*, if it has (subjectively) ”*many* layers” [4]. In the following we adopt the notation of [5] combined with [1]. Neural Networks (for classification) map an input $\boldsymbol{x}$ to a category y approximating a true mapping $y = f^*(\boldsymbol{x})$. A feedforward network defines the parameterized function $y = f(\boldsymbol{x}, \boldsymbol{W})$. Internally, f consists of vector functions $f^{(1)}, f^{(2)}, \dots, f^{(n)}$ with $f(\boldsymbol{x}) = f^{(n)}\left(\dots\left(f^{(2)}\left(f^{(1)}\left(\boldsymbol{x}\right)\right)\right)\dots\right)$, each corresponding to a layer of the network [1].

Let the superscript (n) denote the currently considered layer. The standard, *fully connected*, layer consists of $M_{(n)}$ neurons $z_j^{(n)}$, each connected to every vector component $f_i^{(n-1)}$ of the previous layer's output [1]. We can write layer (n) as vector of the neurons it contains and the output of the j-th neuron of layer (n) as [5]:

$$f^{(n)} = \left(z_1^{(n)}, z_2^{(n)}, \dots, z_{M_{(n)}}^{(n)}\right)^\top \text{ and } z_j^{(n)} = \phi\left(\sum_{i=1}^{M_{(n-1)}} \left(w_{ji}^{(n)} f_i^{(n-1)}\right) + w_{j0}^{(n)}\right) \tag{2.1}$$

where the parameters $w_{ji}^{(n)}$ represent the *weights* and the constants $w_{j0}^{(n)}$ are called *biases*. Both are part of the parameter set $\boldsymbol{W}$.

The function ϕ denotes the neuron's *activation function*. In principle, any differentiable function can be used [5]. Sigmoidal functions like $\sigma(x) = \frac{1}{1+exp(-x)}$ or $tanh^{-1}(x)$ used to be a popular choice [4, 5, 6]. These functions have very small gradients across most of their domain. The resulting insensitivity to their input makes training them via gradients difficult [1]. The *rectified linear unit* (ReLU) $y = max\{0, x\}$ is an easier to optimize activation function, since it preserves most linear properties. Its main disadvantage, 0-gradient for $x < 0$, is mitigated by fixing a non-zero slope γ instead. The resulting *leaky* ReLU $y = max\{0, x\} + \gamma min\{0, x\}$ receives gradient everywhere [1].

2.1.2 Convolutional Layer

The *convolutional layer* is a specialized layer and a building block for the *Convolutional Neural Networks* (CNNs). It uses convolution in place of the general matrix multiplication of fully-connected layers [1].

A typical convolutional layer consists of three stages [1]: In the *convolution* stage, the layer runs convolutions over the input, producing linear activations. The *detector* stage, runs each linear activation through a nonlinear activation function. Finally, the nonlinear activations are fed into a *pooling function*. The pooling stage can also be regarded as its own layer, separate from the convolution [7].

In general, *convolution* is an operation on two functions I, K, defined by [1]:

$$S(t) = (I * K)(t) = \int I(a) K(t-a)\, da \tag{2.2}$$

In the context of convolutional networks, the first argument I is called *input* and the second function K is the *kernel* [1]. If I and K are defined only on integer t, the integral is replaced by a sum in the *discrete* convolution. Discrete or continuous, we can use convolutions also over more than one axis t at a time. If we use a 2D image I as input with a 2D kernel K, we obtain a two-dimensional discrete convolution [1]:

$$S(i,j) = (I * K)(i,j) = \sum_x \sum_y I(x,y) K(i-x, j-y) \tag{2.3}$$

Color images additionally have at least a *channel* for red, blue and green intensity at each position. Assume that each image is a 3D-tensor and $\boldsymbol{V}_{i,j,k}$ describes the value of channel i at row j and column k. Then let our kernel be a 4D-tensor with $\boldsymbol{K}_{i,j,k,l}$ denoting the connection strength (weight) between a unit in input channel j and output channel i at an offset of k rows and l columns between input and output. We may additionally want to sample only every s positions to save computations. With a *stride* s, we obtain the convolution [1]:

$$Z_{i,j,k} = c\left(\boldsymbol{K}, \boldsymbol{V}, s\right)_{i,j,k} = \sum_{l,m,n} \boldsymbol{V}_{l,(j-1)s+m,(k-1)s+n} \boldsymbol{K}_{i,l,m,n} \tag{2.4}$$

Since the kernel $\boldsymbol{K}$ does not depend on the position j, k, we speak of *parameter sharing* [1]. The same structure of neighboring pixels, like an edge, can then be detected anywhere in the image. If we want to extract different features at different locations, we can use the *locally connected* layer [1]. It is related to convolution, but its kernel weights depend additionally on the position j and k. This means the weights are not *shared*, and $\boldsymbol{K}$ is replaced by the 6D tensor $\boldsymbol{K}_{i,j,k,l,m,n}$ in equation 2.4.

The pooling stage produces a summary statistic about nearby outputs of the previous stage. This makes the representation *invariant* to small local translations [1]. A common example is *max pooling* that, given a size $n \times n$, outputs the maximum activation out of each $n \times n$ block of the input.

Padding adds additional zero rows and columns around the input. There are two main types of padding [1]. *Same* padding adds just enough zeros that the output has the same dimension as the input. *Full* padding adds even more zeros - enough to make every input pixel be visited the same amount of times. Convolution or pooling without padding is sometimes referred to as *valid* [1].

2.1.3 Gradient-Based Optimization

In supervised learning, the focus of this work, we are given N inputs $\boldsymbol{x}_i$ and N corresponding target outputs y_i, the model function $f(\boldsymbol{x}, \boldsymbol{W})$ and its parameter set $\boldsymbol{W}$.

Optimization algorithms for training Neural Networks are different from traditional optimization algorithms. Function minimization is the main goal in pure optimization. But in machine learning, we care about some (often intractable) performance measure P. In the hope of improving P, we define a different *cost* or *loss* function $L\left(\boldsymbol{W}\right)$ to be minimized instead [1]. Common examples for L include the *sum-of-squares* function [5]:

$$L\left(\boldsymbol{W}\right) = \frac{1}{2}\sum_{i=1}^{N} \left\| f\left(\boldsymbol{x_i}, \boldsymbol{W}\right) - y_i \right\|^2 \tag{2.5}$$

and the *cross-entropy* (for K classes) between training data and model distribution [5]:

$$L\left(\boldsymbol{W}\right) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ki} \ln f_k\left(\boldsymbol{x_i}, \boldsymbol{W}\right) \tag{2.6}$$

where, for the latter loss, the variables y_i have a 1-of-K encoding indicating the class. The cross-entropy loss led to greater performance in models with sigmoid and softmax outputs, which suffered from slow learning with the squared error loss [1].

The main task of optimization in network training is to minimize the chosen $L(\boldsymbol{W})$ by finding the right parameters $\boldsymbol{W}$. Optimization algorithms used in training are iterative. Starting at a point $\boldsymbol{W}_0$, they generate a sequence of sets $\{\boldsymbol{W}_k\}_{k\in\mathbb{N}}$ until a stopping criterion has been fulfilled [8]. Since we are interested in minimizing L, we stop when approximating a local minimum, where the gradient is (close to) zero. All commonly used algorithms are *line search* [8] methods. For each step k they first compute a search direction p_k and then decide how far to move along p_k. We get $\boldsymbol{W}_{k+1} = \boldsymbol{W}_k + \alpha_k p_k$, where α is called the *step length* [8] or *learning rate* (lr) [5]. Algorithm 1 shows the iterative structure that the main training algorithms follow.

Algorithm 1 Basic line search procedure for network training

```
Require: $L(\boldsymbol{W})$:   The chosen loss function with parameters $\boldsymbol{W}$
Require: $\boldsymbol{W}_0$:   Starting point
1: $k \leftarrow 0$
2: while $\boldsymbol{W}_k$ not converged do
3:     $k \leftarrow k+1$
4:     $p_k \leftarrow p$   Compute current step direction. Gradient descent: $p = -\nabla L(\boldsymbol{W}_{k-1})$
5:     $\alpha_k \leftarrow \alpha$   Compute or use a given step size.
6:     $\boldsymbol{W}_k \leftarrow \boldsymbol{W}_{k-1} + \alpha_k p_k$
7: end while
```

For most algorithms, p_k has to be a descent direction [8], i.e. $p_k^\top \nabla L(\boldsymbol{W}_k) < 0$. The *gradient descent* algorithm employs a simple choice: always move along the steepest descent $p_k = -\nabla L(\boldsymbol{W}_k)$. While modern algorithms add further enhancements, the gradient remains the main information they rely upon.

Error Backpropagation

Modern deep learning models often have millions of free parameters, all of which requiring their gradient in every training step. It is thus essential to compute the gradient efficiently. Fortunately, we can exploit the network's structure by using *backpropagation* [6]. Backpropagation (or BP) uses repeated application of the chain rule to compute the derivatives. It is a special case of reverse mode *automatic differentiation* (AD) [9].

In AD (and by extension in BP) there are two phases [9]: In the *forward* pass, the function is run over the input and intermediate values and their dependencies are recorded. The derivatives are then computed in the second pass. Gradient information is propagated in *reverse* from the outputs to the inputs. This technique is very efficient for functions $f : \mathbb{R}^m \to \mathbb{R}^n$ with $m \gg n$, as the number of passes scales only with n [9]. Since in network training, we optimize a scalar loss value, we only need $O(1)$ passes through the function. According to [9], the constant c hidden by the O-notation is guaranteed to be $c < 6$ and typically $c \sim [2, 3]$.

2.1.4 Challenges in Neural Network Training

Optimization in general is extremely difficult. In this subsection we describe some of the most important challenges more specific to training deep neural networks.

In training, we are confronted with a highly non-convex loss function. This is in part due to structural properties of the model, such as the *weight space symmetry* [1, 5]: In any layer we can swap the incoming weight vector for unit i with the one for unit j and do the same to the outputs. For m layers with n units each, there are $n!^m$ ways to rearrange the units. For certain units like ReLUs, we can scale the incoming weights and biases by some factor ϵ and all outgoing weights by $\frac{1}{\epsilon}$. These examples show that for any local minimum, there is a very large number (possibly uncountably many) of equivalent local minima. This problem is an active area of research nowadays. In practice, it is suspected that for large models most local minima have a low loss value. Finding a global minimum might also not be as important as simply finding a non-optimal point with low loss [1].

For high-dimensional problems, among the points with zero gradient, *saddle points* are more common. While the Hessian is positive definite at a minimum, saddle points have a Hessian matrix with both positive and negative eigenvalues. For many classes of random functions $f : \mathbb{R}^N \to \mathbb{R}$, the ratio of saddle points to minima grows exponentially with N [1]. First-order optimization algorithms are designed to move ”downhill”, so they can escape most saddle points. But Newton’s method might get stuck as it solves for any point where the gradient is zero [1].

In recurrent or very deep neural networks the gradient itself might be problematic. If several small or large weights are multiplied together, the respective partial derivatives might shrink or grow exponentially. In the former case we speak of *vanishing*, in the latter of *exploding* gradients [2]. In practice vanishing/exploding gradients can cause feedforward networks to hardly optimize entire layers at all [2]. The optimization algorithm could also lose progress in a step, if a large gradient causes the parameters to be moved extremely far away [1]. Per-parameter learning rate algorithms and approximate newton methods have a better chance of correcting these problems than pure gradient descent [2]. Careful initialization methods, like *layer-wise pre-training* [10], can help by choosing initial parameters such that issues like vanishing gradients are not an immediate problem [2].

2.2 First-Order Optimization Algorithms

Consistent with literature, we call algorithms using only gradient information *first-order* optimization algorithms. Methods that also use the (approximate) Hessian matrix are called *second-order* optimization algorithms [1, 8].

2.2.1 Stochastic Gradient Descent

In 2.1.3 we introduced gradient descent, which follows the gradient over an entire training set. This is referred to as *batch* learning [5] and can get increasingly expensive for larger training sets. Training can be accelerated by randomly sampling in each step a small subset of points, called *mini-batch*, instead of the whole dataset. We call the number of points in each mini-batch the *batch size* and algorithms using mini-batches are referred to as *stochastic* [1, 11]. Performing gradient descent using mini-batches and a given, not computed, learning rate results in *stochastic gradient descent* (SGD) [12].

There are good reasons beyond computational efficiency to prefer mini-batches over the entire set in training steps. The most important reason is that per-step computation time only grows with the batch size, independent of the size of the training set [1]. In practice, the data in our training set will contain some redundancy. In an extreme example, consider doubling a training set by inserting each sample twice. Then batch learning will require twice the computational effort, while stochastic methods stay unaffected [11]. While such an example is highly unlikely in practice, there may be a large number of examples with similar contributions to the gradient [1]. By only approximating the gradients, a certain amount of "noise" is added, which can help stochastic methods to escape points with zero-gradient [11].

The choice of batch size is influenced by several factors [1] (list was also in the Seminar-Paper, see 2):

- A larger batch size provides a more accurate estimate, but with sublinear returns. If we estimate the mean of n examples, the standard error is given by $\frac{\sigma}{\sqrt{n}}$ where σ is the true standard deviation of the sample values [1]. A 100 times larger batch leads to 100 times more computation, but only a 10 times more accurate gradient.
- Too small batches underutilize parallel capabilities of modern hardware. Below a minimum size, no speedup in processing each batch is achieved.
- Certain hardware, like GPUs, work best with specific sizes, typically powers of 2.
- The main limiting factor: Memory requirements scale linearly with the batch size.

Choosing a suitable learning rate is much harder. Typically, the rate α_k is gradually decayed from a chosen initial value α_0, staying constant after a certain number of iterations [1]. Many different learning rate schedules have been proposed, like the more complex *cyclical learning rate* schedule in [13]. Convergence of SGD is guaranteed, if the learning rate sequence $\{\alpha_k\}_{k\in\mathbb{N}}$ fulfills the Robbins-Monro conditions [11]:

$$\sum_{k=1}^{\infty} \alpha_k = \infty \quad and \quad \sum_{k=1}^{\infty} \alpha_k^2 < \infty \tag{2.7}$$

The choice of the learning rate greatly influences training performance: Too low and the learning will progress slowly or even get stuck at a high loss-value plateau. Too high and we might oscillate in progress or even diverge entirely [1].

The main advantage of SGD is quick initial progress, even in large Datasets. While it can be improved via *momentum* [14] or *Nesterov momentum* [15], it is often outclassed by the newer *adaptive learning rate* algorithms. They keep a separate learning rate for each parameter and adapt these rates throughout training [1].

2.2.2 AdaGrad

Adaptive learning rate algorithms can be seen as approximating the so-called empirical Fisher matrix [2]. The empirical Fisher itself somewhat approximates second-order curvature information [2]. In the case of *AdaGrad* [16], its approximator matrix G_t is the sum of the outer products of all t past gradients:

$$G_t = \sum_{i=1}^{t} g_i g_i^\top \tag{2.8}$$

with $g_i = \nabla L\left(\boldsymbol{W}_i\right)$ [16].

The learning rates are scaled inversely proportional to the square root of the sum of all historical squared values. Note that $diag(G_t) = \sum_{i=1}^{t} g_i \circ g_i$ is precisely this sum (with $\circ$ denoting the *Hadamard*, or element-wise, product). Adding a small constant $\delta \approx 10^{-7}$ to avoid division by zero, this yields the weight update:

$$\boldsymbol{W}_{k+1} = \boldsymbol{W}_k - \frac{\alpha_k}{\delta I + \sqrt{diag\left(G_t\right)}} \circ \nabla L\left(\boldsymbol{W}_k\right) \tag{2.9}$$

We can improve efficiency by directly computing the diagonal of G_t. For this, we accumulate the current sum in r_k and add the new squared gradients in each step: $r_{k+1} = r_k + \nabla L\left(\boldsymbol{W}_k\right) \circ L\left(\boldsymbol{W}_k\right)$ with $r_0 = 0$. The final weight update then becomes:

$$\boldsymbol{W}_{k+1} = \boldsymbol{W}_k - \frac{\alpha_k}{\delta + \sqrt{r_{k+1}}} \circ \nabla L\left(\boldsymbol{W}_k\right) \tag{2.10}$$

Using the simplified notation from [1], this derivation was a very short summary of the work in [16].

Since the learning rates are scaled inversely proportional to the sum-of-gradients, parameters with larger partial derivatives rare decreased the fastest, while parameters with small derivatives have a slower decrease in learning rate. As a result, AdaGrad can make greater progress on surface regions with gentler slopes [1]. Originally developed for convex optimization, it performs well for some but not all types of deep networks. Accumulating *all* gradients can cause the learning rate to decrease too quickly [1].

2.2.3 RMSProp

RMSProp [17] improves on AdaGrad by changing the gradient accumulation into an exponentially weighted moving average [1]. The use of the moving average introduces a decay rate ρ that controls how many gradients are considered in each update. The RMSProp update step works exactly like AdaGrad, except for the accumulator r_k:

$$r_{k+1} = \rho r_k + (1-\rho)\,\nabla L(\boldsymbol{W}_k) \circ L(\boldsymbol{W}_k) \tag{2.11}$$

AdaGrad converges quickly when applied to a convex function. For a non-convex function training a neural network, we may pass many different structures and finally arrive at a locally convex region [1]. Old historical entries may have caused AdaGrad to shrink the learning rate too much before reaching at a convex region. Thanks to the decaying average, RMSProp discards old information, avoiding such slowdowns. Inside a convex structure, it converges just like an instance of AdaGrad initialized within the structure [1]. RMSProp is considered "one of the go-to methods" [1] used for training, due to its effectiveness at optimizing deep neural networks.

2.2.4 Adam

Of the three presented adaptive learning rate algorithms, *Adam* [18] (short for adaptive moment estimation) is the most sophisticated. In addition to r_k, it employs an exponential moving average s_k of the gradients (not squared). Both have their own decay rate: β_1 for s_k and β_2 for r_k. In [18], s_k is considered an estimate of the 1^{st} moment (mean) and r_k an estimate of the 2^{nd} raw moment ("uncentered variance") of the gradient. Their initialization as vectors of 0's lead to moment estimates "biased towards zero" [18]. Adam corrects these biases with the estimates $\widehat{s_k} = s_k/\left(1-\beta_1^k\right)$ and $\widehat{r_k} = r_k/\left(1-\beta_2^k\right)$. All pieces put together, this results in the following procedure [1, 18]:

Algorithm 2 Adam

```
Require: $\alpha_0$:   Initial learning rate (and a schedule to compute $\alpha_k$)
Require: $\beta_1, \beta_2 \in [0,1)$:   Exponential decay rates
Require: $L(\boldsymbol{W})$:   Loss function with parameters W
Require: $\boldsymbol{W}_0$:   Starting point
 1: $s_0, r_0 \leftarrow 0$   (Initialize moment estimates)
 2: $k \leftarrow 0$
 3: while $\boldsymbol{W}_k$ not converged do
 4:     $k \leftarrow k+1$
 5:     $\alpha_k \leftarrow \alpha$   (Compute or use given learning rate)
 6:     $g_k \leftarrow \nabla L(\boldsymbol{w}_{k-1})$   (Gradient at current step)
 7:     $s_k \leftarrow \beta_1 s_{k-1} + (1-\beta_1)\,g_k$   (Biased $1^{st}$ moment)
 8:     $\widehat{s_k} \leftarrow \frac{s_k}{(1-\beta_1^k)}$   (Correct bias in $1^{st}$ moment)
 9:     $r_k \leftarrow \beta_2 r_{k-1} + (1-\beta_2)\,(g_k \circ g_k)$   (Biased $2^{nd}$ moment)
10:     $\widehat{r_k} \leftarrow \frac{r_k}{(1-\beta_2^k)}$   (Correct bias in $2^{nd}$ moment)
11:     $\boldsymbol{W}_k \leftarrow \boldsymbol{W}_{k-1} - \alpha_k \frac{\widehat{s_k}}{(\delta+\sqrt{\widehat{r_k}})}$   (Update the weights)
12: end while
```

Compared to Adam, RMSProp lacks the bias-correction term for the accumulated gradient moments. In case of β_2 (ρ for RMSProp) close to 1, this leads to large stepsizes and possibly even divergence [18]. Adam is considered robust to the choice of hyper-parameters [1] and it is invariant to gradient rescaling [18]. In general, Adam combines advantages from RMSProp and AdaGrad. In many scenarios, Adam performs at least as well as RMSProp. For sparse gradients, Adam matches AdaGrad, both outperforming RMSProp [18].

2.3 Second-Order Optimization

Challenges in network learning can be seen as a special case of problems arising in general non-linear optimization. For example, parameters are often tightly coupled and have strong local dependencies. Different directions in parameter space might have large variations in scale. Gradient descent is very sensitive to these issues and must drastically lower its learning rate to avoid instability in these situations [2].

Second-order optimization methods are better equipped to solve the problem of scale and curvature variations along different directions. They rescale the gradient along the different "eigen-directions" of the *curvature matrix* B according to their associated eigenvalue (curvature) [2].

2.3.1 Newton's Method

The classic second-order method is the *Newton-Raphson*, or simply *Newton's* method, with all second-order methods deriving from it [2].

Around the current point $\boldsymbol{W}_k$, we approximate the function $L(\boldsymbol{W}_k + \delta)$ by a local quadratic model using the curvature matrix B [2]:

$$L(\boldsymbol{W}_k + \delta) \approx \frac{1}{2}\delta^\top B_k \delta + \nabla L(\boldsymbol{W}_k)^\top \delta + L(\boldsymbol{W}_k) \tag{2.12}$$

For the standard Newton's method, B_k is given by the Hessian H and the quadratic model becomes a second-order Taylor series expansion ignoring higher derivatives [8]. During the update step we have to solve the linear system $B_k\delta = -\nabla L(\boldsymbol{W}_k)$, called the Newton equation. This yields the newton step:

$$\boldsymbol{W}_{k+1} = \boldsymbol{W}_k - B_k^{-1}\nabla L(\boldsymbol{W}_k) \tag{2.13}$$

If we are in a neighborhood of the solution $\boldsymbol{W}^*$ and the Hessian is positive semidefinite, then Newton's method converges quadratically towards $\boldsymbol{W}^*$ [8].

Unmodified Newton's method might run into problems when used for training neural networks, due to the highly non-convex loss function. In non-convex regions, the Hessian is indefinite, causing Newton's method to move in the wrong direction [2]. A way to counteract this issue is to apply *damping* techniques. Arguably the simplest is called *Tikhonov regularization* [19] and adds, for some positive τ, a multiple of the identity matrix τI to H, obtaining $B = (\tau I + H)$ which is again positive definite [8, 2]. If H

has negative eigenvalues close to zero, Tikhonov regularization works fairly well. But for extreme negative curvature, τ would have to be very large, and B dominated by the τI term. Then, Newton's method converges to SGD, with $(1/\tau)$ times the stepsize [1].

Beyond structural problems, the main disadvantage of Newton's method is its computational cost. In a deep network with $\|\boldsymbol{W}\| > 10^7$ parameters, simply computing or storing the Hessian with $\|\boldsymbol{W}\|^2$ entries is impractical. Newton's Method additionally requires inverting the Hessian, in time $O\left(\|\boldsymbol{W}\|^3\right)$, *in every iteration* [1].

2.3.2 Fast exact Multiplication by the Hessian

While the computation of the Hessian remains expensive, we can gain "cheap" access to its curvature information. Specifically, [20] proposed a method to compute the Hessian vector product $H\boldsymbol{v}$ for any $\boldsymbol{v}$ in just two (instead of $\|\boldsymbol{W}\|$) backpropagations.

In [20], the new differential operator $\mathcal{R}\left\{\cdot\right\}$ is introduced. It is defined as:

$$\mathcal{R}_{\boldsymbol{v}}\left\{f\left(\boldsymbol{W}\right)\right\} = \frac{\partial}{\partial r} f\left(\boldsymbol{W} + r\boldsymbol{v}\right)\Big|_{r=0} \tag{2.14}$$

Then simply $H\boldsymbol{v} = \mathcal{R}_{\boldsymbol{v}}\left\{\nabla L\left(\boldsymbol{W}\right)\right\}$. We can compute this in just two backprops. Specifically, for a twice continuously differentiable function f, we obtain:

$$H\boldsymbol{v} = \mathcal{R}_{\boldsymbol{v}}\left\{\nabla L\left(\boldsymbol{W}\right)\right\} = \left(\frac{\partial}{\partial r}\nabla_{\boldsymbol{W}} L\left(\boldsymbol{W} + r\boldsymbol{v}\right)\right)\Big|_{r=0} = \nabla_{\boldsymbol{W}}\left(\nabla L\left(\boldsymbol{W}\right)^\top \boldsymbol{v}\right) \tag{2.15}$$

The proof of the third equality is omitted here, but is included in [3]. The resulting formula is both efficient and numerically stable [20].

2.3.3 The EH-Newton Algorithm

Many approximate second-order algorithms have been developed that try to gain advantages of Newton's method without the computational burden. The most prominent include *nonlinear Conjugate* Gradients and quasi-Newton methods like *(L-)BFGS* [8], and more recently the *Hessian-Free* and *K-FAC* algorithms [2]. In practice none of them have displaced first-order methods as the main algorithm choice. Seeing how especially the latter K-FAC method works much faster than plain gradient descent [2], we suspect that often first-order algorithms are chosen mainly for their ease of implementation. Therefore, we will focus here on a comparatively simple approach.

The main computational bottleneck of Newton's method is solving the system of linear equations $B_k\delta = -\nabla L\left(\boldsymbol{W}_k\right)$. Instead of solving exactly, we can approximate the solution using the (linear) *Conjugate Gradients* method (CG). This approach has already been proposed in [21] and is known as *Truncated Newton*. Originally, it computed the Hessian-vector product via numerical approximation. In [3] the use of the fast exact product (see 2.3.2) is added. To emphasise the use of this product, we call the resulting algorithm *Efficient-Hessian Newton*, or EH-Newton.

In short, CG solves $Ax = b$ by producing a set of directions $\{p_0, p_1, \dots, p_n\}$ conjugate w.r.t. the matrix A. That is, $\forall i \neq j : p_i^\top A p_j = 0$. For a detailed derivation of the updates, see [8]. The final CG method is shown in algorithm 3. Note, that it only requires Matrix vector products, and never needs A itself. Since we solve the newton equation, $A = H$ and by the previous section, we can compute the Hessian-vector products efficiently.

Algorithm 3 Conjugate Gradients

```
Require: $A$, $b$:   We want to solve $Ax = b$ for $x$.
Require: $x_0$:   Initial estimate for $x$.
1: $r_0 \leftarrow Ax_0 - b$, $p_0 \leftarrow -r_0$, $k \leftarrow 0$
2: while $r_k$ too large do
3:     $\alpha_k \leftarrow \frac{r_k^\top r_k}{p-k^\top A p_k}$   Compute step size
4:     $x_{k+1} \leftarrow \alpha_k p_k$   Apply step
5:     $r_{k+1} \leftarrow r_k + \alpha_k A p_k$   Compute new residual
6:     $\beta_{k+1} \leftarrow \frac{r_{k+1}^\top r_{k+1}}{r_k^\top r_k}$   Factor, such that $p_{k-1}$ and $p_k$ are conjugate w.r.t. A
7:     $p_{k+1} \leftarrow -r_{k+1} + \beta_{k+1} p_k$   Compute new step direction
8:     $k \leftarrow k + 1$
9: end while
```

One last precaution is needed. If the gradient vector points in a direction with negative curvature, or simply due to numerical ill-conditioning, the CG-generated solution p_k may not be feasible [21, 3]. Therefore, we check $\nabla L(\boldsymbol{W}_k)^\top p_k$ against a small positive threshold. If larger, feasibility is not guaranteed and we simply revert to using the steepest descent direction [3].

Algorithm 4 shows the resulting procedure. Inside the `CG()`-subroutine, the product $(H + \tau I)\, v$ should be computed as $Hv + \tau v$ using the efficient Hessian-vector product.

Algorithm 4 (Truncated) EH-Newton

```
Require: $L(\boldsymbol{W})$:   The chosen loss function with parameters $\boldsymbol{W}$
Require: $\boldsymbol{W}_0$:   Starting point
Require: $\tau$:   Tikhonov regularization/damping factor
1: $k \leftarrow 0$
2: while $\boldsymbol{W}_k$ not converged do
3:     $k \leftarrow k + 1$
4:     $p_k \leftarrow \texttt{CG}((H + \tau I), -\nabla L(\boldsymbol{W}_k))$   Approx solution to $(H + \tau I)\, p_k = -\nabla L(\boldsymbol{W}_k)$
5:     if $\nabla L(\boldsymbol{W}_k)^\top p_k > \tau$ then $p_k \leftarrow -\nabla L(\boldsymbol{W}_k)$   Feasibility check.
6:     $\alpha_k \leftarrow \alpha$   Compute or use a given step size.
7:     $\boldsymbol{W}_k \leftarrow \boldsymbol{W}_{k-1} + \alpha_k p_k$
8: end while
```

2.4 Modern Convolutional Neural Networks

Convolutional Networks have long played an important role in deep learning. In some ways, they even ”carried the torch for the rest of deep learning” [1]. They were some of the first models to perform well and have been used in commercial applications for decades. For example, by the end of the 1990s, a system based on CNNs developed at AT&T [22] was used to read over 10% of all checks in the US [1].

More recently, deep convolutional networks have risen to prominence, after they have been used in 2012 to win the *ImageNet Large-Scale Visual Recognition Challenge* (ILSVRC) [23]. Since then many architectures have been designed to improve performance even further, or to increase computational efficiency while maintaining reasonable accuracy. In the following, we present the main ideas behind three of the most prominent modern architectures.

2.4.1 Inception Architecture

The *Inception* [7, 24, 25] architecture underlies a series of models. The first, 22-layer deep *GoogLeNet*, won at the ILSVRC in 2014 [7].

The main idea of the Inception architecture, is the *Inception module*. The final network mainly consists of several stacked inception modules. Each module takes as input a 3-D tensor ($height \times width \times channels$) and produces as output another 3-D tensor. Internally, the input is fed into four alternative parallel paths [7]: a path for 1×1 convolutions, one for 3×3 convolutions, a path for 5×5 convolutions[1] and one path with pooling. Here, we refer to convolutions with a $n \times n \times c$ kernel (with c channels) as $n \times n$ convolution. All convolutions in the Inception architecture are followed by ReLU activations [7]. Each path produces a 3-D tensor of the same height and width, varying just the number of channels. The output of all paths is then concatenated, forming the module’s output.

Assuming each unit of the earlier layer (now input) corresponds to some region in the image, then correlated units concentrate in local regions [7]. They can be covered by the layer of 1×1 convolutions in the next module. Clusters that are more spatially spread out are covered by the larger 3×3 and 5×5 convolution layers of the next module. The larger convolutions, especially the 5×5, are computationally expensive. To reduce their costs, a 1×1 convolution with fewer channels is added before all 3×3 and 5×5 convolutions. This is called *dimensionality reduction* [7]. Figure 2.1 shows the final Inception module with its 4 paths and dimension reduction (yellow) applied to the larger convolutions.

In addition to a main final layer of logits, Inception networks have *auxiliary classifiers* [7]. These extra layers are built on top of earlier layers in the model. Originally thought to help with gradient propagation, they were found to have a regularizing effect [24]. Inception-v3 and onwards also *batch-normalize* [26] the auxiliary classifier.

[1]Later versions replaced 5×5 convolutions by two layers of 3×3 convolution [24].

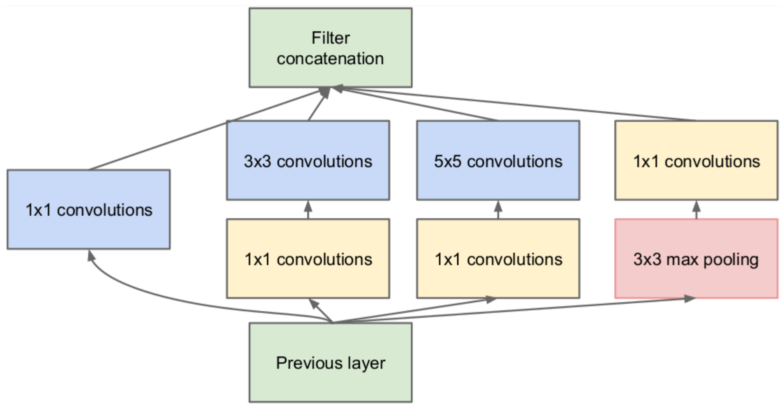


Figure 2.1: Inception module with dimensionality reduction
From Figure 2(b) in [7]. © 2015 IEEE

This architecture brings certain advantages [7]: It allows for building stages with many more units without blowing up the computational complexity. The parallel structures within the modules align with the intuitive idea of processing visual information at various scales. The aggregation of outputs enables the next module to abstract features from all scales simultaneously.

2.4.2 Residual Networks

A *deep residual learning* framework is introduced in [27]. As in the paper, let $\mathcal{H}(\boldsymbol{x})$ denote the desired underlying mapping to be learned by the model. The main idea of residual learning is to learn $\mathcal{F}(\boldsymbol{x}) := \mathcal{H}(\boldsymbol{x}) - \boldsymbol{x}$ instead. The original mapping is then equal to $\mathcal{F}(\boldsymbol{x}) + \boldsymbol{x}$, which is supposed to be easier to learn optimize than the original mapping $\mathcal{H}$. The new formulation $\mathcal{F}(\boldsymbol{x})+\boldsymbol{x}$ can be realized by adding ”shortcut connections” [27]. The modified layer then takes as input not just the previous layer, but also the output from two or more layers behind. Figure 2.2 shows this modification. The shortcut identity introduces no new parameters to be optimized.

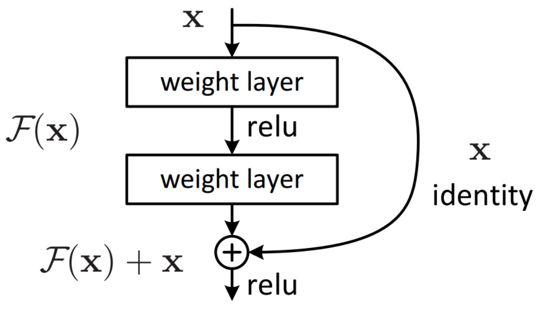


Figure 2.2: Residual learning: a building block.
From Figure 2 in [27]. © 2016 IEEE

This block forms the basis of *Residual Networks* (ResNet). Deeper blocks (skipping three layers) are also used in the deepest ResNet models. As is shown in Figure 2.2, ReLU units serve as layer nonlinearities. The weight layers are mostly 3×3 convolutions with the number of channels growing inversely proportional to the output size, to keep the same time complexity in each layer [27]. Batch-norm is applied right after each convolution. Output size is halved several times in the network, by using stride 2 convolutions. Whenever this happens (and the number of channels doubles), the identity shortcut can either be padded with zeros, or be replaced by a 1×1 convolution (introducing extra parameters) [27].

This reformulation seems to ”provide reasonable preconditioning” [27] when learning the underlying mapping. The new shortcut connections counteract the phenomenon of *degradation*, where deeper models surpassing a certain depth have again larger training and test errors. This allows much deeper networks to be trained effectively, like the 101-layer ResNet, or even a 152 layer variant. An ensemble of the latter won at the ILSVRC 2015 with a top-5 error of only 3.57% [27].

2.4.3 MobileNet

MobileNets [28, 29, 30] are a group of models tailored for mobile and low-resources environments.

The core of MobileNets are *depthwise separable convolution* layers [28]. They separate standard convolution into a depthwise and a pointwise layer. The depthwise layer applies one filter each per input channel. To combine the channels into new features, a 1×1 (or pointwise) convolution is used. While reducing accuracy slightly, it is extremely efficient. For example, a 3×3 depthwise separable convolution uses 8 to 9 times less computation than a standard convolution [28]. Batchnorm and ReLU activations are used after both layers.

Version 2 introduced the *inverted residual with linear bottleneck* module [29]. It starts with a 1×1 convolution, expanding the number of channels by a factor greater than 1. According to [29], this is in contrast to residual blocks, where the factor is smaller than 1. This is followed by a depth-wise 3×3 convolution (possibly strided). A second 1×1 convolution is then used to project the output into a dimension with less-than-expanded channels. This second convolution has a linear activation, as it was found to increase performance [29]. If and only if input of the first 1×1 conv. and output of the second 1×1 conv. have the same number of channels, a residual connection (”shortcut”) is added between input and output [30].

MobilenetV3 is mainly a new combination of these layers as building blocks. It was tuned automatically by *hardware-aware network architecture search* (NAS) in combination with the NetAdapt algorithm [30].

All MobileNets also include two hyperparameters to trade-off accuracy and performance [28]: the *width-multiplier* α and the *resolution-multiplier* ρ. The number of input and output channels of each layer is simply multiplied by α. The input image (and thus implicitly every subsequent layer) is scaled by ρ.

3 Implementation

This chapter describes in detail how the second-order algorithm was integrated with the TensorFlow [31] library. Additionally, we list the computational setup used in the following chapter to compare the algorithms' training performance.

Specifically, we implemented a new `tf.Optimizer` executing a step of the algorithm in its update step. Using this integrated optimizer we can leverage the higher-level APIs provided by TensorFlow, such as Keras and TensorFlow Slim. These APIs (described later) simplify the declaration and training of large-scale deep neural networks. In particular, we use TF Slim to declare and train our convolutional networks.

3.1 Tensorflow

In the 2015 white paper, TensorFlow is introduced as both an "interface for expressing machine learning algorithms" as well as the "implementation for executing such algorithms" [31].

The TensorFlow programming model consists of two main steps[1]:

- Define computations in form of a "stateful dataflow graph" [31].
- Execute this graph, which now remains fixed, with possibly different inputs.

The *define-and-run* [9] approach enables optimizations of the graph structure, at the expense of limited control flow and being more difficult to debug.

3.1.1 Computational Graph Model

The dataflow graph mentioned above consists of *nodes* and *edges*. Nodes represent *operations* (ops), such as matrix multiplication, while edges indicate dependencies between nodes [31]. The input of an operation is represented by zero or more incoming edges into its graph node. Zero or more outgoing edges provide the outputs. Along these edges, values flow in form of *tensors*, arrays of arbitrary dimension. The shape and underlying element type of a tensor can be specified explicitly or inferred while constructing the graph [31]. Figure 3.1 shows a small working example. Note that lines 3 to 8 don't compute any result — they merely add op nodes to the graph.

[1]Later TensorFlow versions (1.7 onwards) additionally include *eager execution* capabilities. In this mode, the native Python interpreter is used to immediately evaluate operations, without first building a graph. However, due to a possible performance overhead without functional benefits, we do not make use of this functionality in this thesis.

```python
import tensorflow as tf

A = tf.constant([[1., 2.],
                 [4., 5.]], name="A")
b = tf.ones([2, 1], name="b")
x = tf.constant([[10.], [10.]], name="x")
o = tf.add(tf.matmul(A, x, name="Ax"), b,
           name="result")
with tf.Session() as sess:
  res = sess.run(o) # executes the graph
  print(res) # res is a python value
```

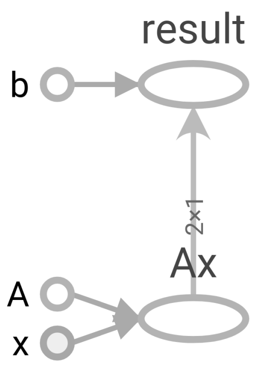


Figure 3.1: Exemplary TensorFlow code and corresponding computational graph

After a graph is created, it can be executed within a *Session*. Upon executing its `run()` method, TensorFlow computes the transitive closure of nodes needed to produce the requested output, and arranges their order respecting dependencies [31]. This op rescheduling aims to minimize the time during which intermediate results are kept in memory. As a result, peak memory consumption can be reduced – important for GPUs with limited memory or highly contended network connections [31].

Most regular tensors are only stored during one execution of the graph. A *Variable* is a mutable tensor that stores state across executions of the graph [31]. Stateful operations, such as `assign` or `assign_add` can change the value of the variable. Handles to the mutable tensor can also be passed to regular tensor operations.

Machine learning models typically store their parameters in variables [31], each roughly corresponding to a layer of the model. An *Optimizer* can be used to compute variable updates during training. This happens in two steps: `compute_gradients()` computes the gradients of the loss w.r.t. the trained variables and `apply_gradients()` applies one step of the optimization algorithm using the computed gradients.

Building the graph out of simple ops, like additions or multiplications, is possible but error-prone and not always intuitive. To train the model on a dataset, the built graph is usually executed over many `Session.run` calls, forming the *training-loop*. Both of these steps are handled automatically by higher-level abstractions also included in TensorFlow. Two such APIs are Keras [32] and (in TensorFlow 1) TF-Slim [33].

3.1.2 Keras

Keras is a high-level deep learning API, running on top of TensorFlow[2]. It is packaged with TensorFlow as the module `tf.keras`. Keras aims to simplify declaring and training ML models. The core abstractions of Keras are *layers* and *models* [32].

A layer consists of a *tensor-in tensor-out* [32] computation function and state, stored

[2]Multi-backend Keras is superseded by `tf.keras` according to the README on the Keras github page. (https://github.com/keras-team/keras, accessed on 17-Jun-2020)

in TensorFlow variables. A wide variety of layers come built-in with Keras, including core layers (e.g. the dense fully-connected), activation layers (such as ReLU), convolution layers, pooling layers and even recurrent layers or regularization and normalization layers [32]. Custom layers can be defined by subclassing the base Layer class. The internal state is composed of the two TF variable lists `trainable_weights` and `non_trainable_weights`, the former being included in backprop. Every layer is a callable and the logic of applying the layer to the input tensors is performed in `call()` [32].

Models are objects representing groups of layers with added training and inference features [32]. They consist of one or more `inputs` and one or more `outputs` connected by the model's `layers`. Like layers, models are callables and the logic of applying the layers to the inputs is performed in `call()`. These calls can be chained together like layers or even nested (model consisting of multiple models) [32].

Apart from subclassing the base Model class, the main ways to instantiate models are *Sequential* and the *Functional API* [32]. A Sequential model represent a list of layers where each layer has exactly one input and one output tensor. While convenient, Sequential models have certain limitations, most importantly non-linear network topology (e.g. a residual connection) is not supported [32].

The functional API is a more flexible way to create models: as *graphs of layers*. Layers represent nodes and their `call` methods add an edge to the graph [32]. For example, if `a` and `b` are layers then calling `out=a(b)` corresponds to a graph with nodes {`a,b`} and an edge from `a` to `b`. The same graph of layers can be used to define multiple models and each model can have multiple inputs and/or outputs [32].

As mentioned above, model training is simplified in Keras. Given a declared model, it consists of two method calls: `model.compile` and `model.fit` [32]. Within compile, the loss function is specified, as well as the optimizer. The fit method trains the model, using its declared loss and optimizer, for a fixed number of epochs.

3.1.3 Tensorflow-Slim

TF-Slim is a high-level library on top of TensorFlow. In many ways similar to Keras, it is – as the name suggests – more lightweight. Slim also introduces the *layer* abstraction, new *variables* and *argument scoping*. Additionally, Slim includes common regularizers, losses, metrics and many widely used convolutional neural network models [33]. Most functions in TF-Slim are thin wrappers around groups of TensorFlow operations and aim to reduce code clutter and speed up model development.

TF-Slim layers, like their Keras counterpart, are higher-level concepts that consist of several TensorFlow operations and can store parameters in variables [33]. They are also callable and can be called with tensors or other Slim layers as inputs. While less extensive than Keras, TF-Slim provides several built-in layers, including convolutional, fully connected, dropout and batch norm [33]. Via a single `repeat` or `stack` method call, a sequence of multiple layers can be declared. It is more limited than Keras' Sequential Model, however, as the Slim methods can only sequence a single layer type.

Variables in Slim are thin wrappers around TensorFlow variables, declaring shape, the initializer, regularization and device placement in one method call [33]. Slim divides its variables in two categories: *model variables* and *regular variables*. Model variables represent parameters of a machine learning model. They are updated during learning and possibly loaded from a checkpoint during inference. Layers define model vars and plain TF vars can be added to the model var collection. All other variables are called regular variables and not stored into or loaded from checkpoints [33].

The *argument scoping* mechanism adds unique functionality to Slim. The scope is declared with a set of operations and a set of arguments. Within the scope, each op contained in the scope's op-set will receive the additional arguments from the scope's arg-set [33].

TF-Slim provides functionality for training and evaluating models. A single call to `slim.learning.train` starts a training loop that updates the model, logs model statistics such as the loss, and periodically saves the model in a checkpoint [33]. The stored model can be completely or partially restored from the checkpoint using one of several `slim.get_variables_*` helper methods. Training models can be evaluated in parallel along defined metrics using `slim.evaluation.evaluation_loop` [33].

The separately offered *TensorFlow-Slim image classification model library* builds on top of the core TF-Slim library [34]. It includes several CNNs like Inception, ResNet or MobileNet built with Slim layers. Finally, it also provides two python scripts for training and inference. These scripts handle model instantiation, model deployment, handling datasets (like Imagenet) and running the training or inference automatically. They can be configured via flags to use the desired models, optimizers, datasets and hyperparameters.

This second library was the main reason why TF-Slim was chosen over Keras for our thesis. The `train_image_classifier.py` and `eval_image_classifier.py` scripts were modified to use the local copy of `tf_slim` instead of `tf.contrib.slim` (and to include our new optimizer next to the built-in TF optimizers). We also had to change `optimizer_device()` in `deployment/model_deploy.py`, a helper file used by the training script. The function now places the optimizer on a GPU, which is necessary for our `EHNewton` optimizer to execute ops on the GPU (see section 3.2). Regular TF optimizers are not affected by this new placement.

3.2 Integrating the EH-Newton Algorithm into TensorFlow

At the heart of model training in TensorFlow lies the `Optimizer`. In TF 1 there are two sets of optimizers.One set is packaged in `tf.train` and the other set comes with Keras, exported as `tf.keras.optimizers`. Both sets have a base optimizer class (in `tf.train.Optimizer` or in `tf.python.keras.optimizer_v2.Optimizer_v2`) that is subclassed by different optimizer algorithms (like Adam or SGD).

The base class handles the two main steps of optimization: `compute_gradients()` and `apply_gradients()`. When applying the gradients, for each Variable that is optimized, the method `_resource_compute_dense(grad, var)`[3] is called with the variable and its (earlier computed) gradient. In this method the algorithm update step for this variable is computed. It has to be overridden by any subclassing optimizer.

For this thesis, we implemented two versions of our optimizer: one inheriting from the Optimizer in `tf.train` and one inheriting from the Keras Optimizer_v2. Because they are functionally equivalent, we will focus here on the `tf.train.Optimizer` subclass `EHNewtonOptimizer`. Both versions loosely build on the implementation of Julian Suk in his Master's thesis [3].

Like with any other TF optimizer, the user has to instantiate an object before using *EHNewton*. The constructor accepts the *learning rate* as well as the EHNewton hyperparameters: the regularization factor τ, the *CG-convergence-tolerance* and the *max. number of CG-iterations* (see 2.3.3). Internally, the parameters are converted to tensors and stored as python object attributes.

The main logic happens in the above mentioned *_resource_compute_dense(grad, var)* method. It works in three steps:

- Compute unscaled step: `step = self._newton_step(grad, var.handle)`
- Scale the step: `scaled = math_ops.multiply(step, self._lr_t)`
- Assign: `return state_ops.assign_add(var, scaled, ...)`

The `_newton_step()` method first computes an approximate CG-solution x to

$$(H + \tau I)x = -\nabla_{var} f \tag{3.1}$$

where H is the Hessian of f w.r.t. *var*, by calling `_cg_solve()`. Then, as suggested in [3], it compares $(\nabla_{var} f)^\top x$ to the threshold $-\tau$. If $(\nabla_{var} f)^\top x$ is larger, then feasibility is not guaranteed and we take a simple gradient descent step instead. To explain the loop body of `_cg_solve()`, we refer to figure 3.2. It shows the code snippet and each equivalent line in pseudocode. The callable `Ax()` is passed to *_cg_solve* from within *newton_step*. It computes the left-hand side of equation (3.1). That loop body is executed at most *maximum_iterations* times inside a `tf.python.ops.while_v2.while_loop`. It is the only control-flow op in TF that supports the necessary second-order derivatives.

3.3 Computational Setup

3.3.1 Hardware

For most of the code development and test runs we used a computer provided by the Chair of Scientific Computing at TUM. The machine had 16 GB of RAM and a four-core

[3]To optimize *sparse variables*, the method `_resource_compute_sparse()` needs to be implemented. There is no algorithmic difference between the dense and the sparse case, but working with sparse variables requires significant implementational effort. Therefore we only consider dense variables in our thesis.

```
# one CG iteration to approximate Ax=b
rtr = self._vv(r, r)  # r is the residual
axp = Ax(p)  # Ax(p) computes A*p
alpha = math_ops.divide(rtr, self._vv(p, axp))
x = math_ops.add(x, math_ops.multiply(p, alpha))
r = math_ops.subtract(r, math_ops.multiply(axp, alpha))
rtr_new = self._vv(r, r)
beta = math_ops.divide(rtr_new, rtr)
p = math_ops.add(r_ret, math_ops.multiply(p, beta))
return x, r, p  # the new values of x, r and p
```

```
Lines 2 to 4:
$\beta_{old} \leftarrow r^\top r$
$\alpha \leftarrow (r^\top r)/(p^\top Ap)$
Line 5: $x \leftarrow x + \alpha p$
Line 6: $r \leftarrow r - \alpha Ap$
Lines 7 to 9:
$p \leftarrow r + p\frac{r^\top r}{\beta_{old}}$
```

Figure 3.2: The while-loop body of `_cg_solve()` and the corresponding pseudocode.

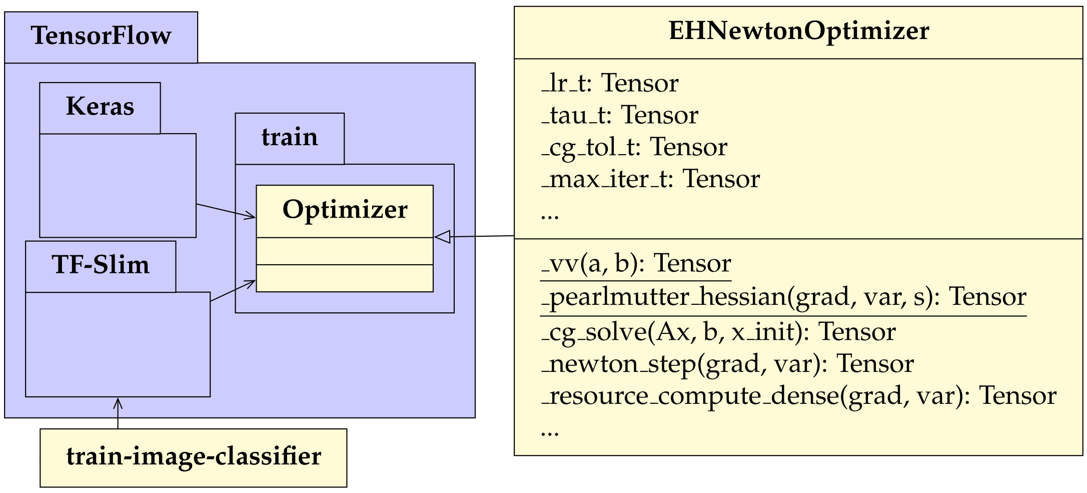


Figure 3.3: UML diagram of the main software components and their relations. All shown attributes and methods are *protected*.

(8 hyper-threaded) Intel Core i7 3770K processor. It had one NVIDIA Titan XP (12GB VRAM) and one Quadro P4000 GPU (8GB VRAM). The Titan GPU was used to train the models, while evaluation could run in parallel on the P4000.

Additionally, near the end of this thesis, we had a few hours of access to a virtual server provided by the Leibniz Supercomputing Centre. The server provided a 16 core Intel Xeon CPU, 240 GB of RAM and a dedicated NVIDIA P100 GPU (16GB VRAM).

3.3.2 Software

The OS on both the computer and the virtual server was Linux Ubuntu. Python `3.7` with TensorFlow version `1.15` (and TF-SLim v. `1.1.0`) was used. Figure 3.3 shows a UML diagram with the relations of the main classes and packages in our project. The *train-image-classifier* class refers to the modified *train_image_classifier.py* script. This diagram is only an overview, as only the most relevant classes and relations are shown.

4 Evaluation

In this section we compare the performance of the presented EH-Newton algorithm to the state-of-the-art first order optimizers. For this, we train one instance each of the three model architectures from section 2.4: InceptionV3, ResNet-50 and MobileNetV2.

4.1 The ImageNet Dataset

For training, we use the ImageNet Large Scale Visual Recognition Challenge dataset [35]. It contains almost 1.3 million images split into 1000 object classes. These classes have been selected out of the set of all ImageNet categories, such that their synsets (synonym sets) don't overlap [35]. The categories are diverse (animals, objects, places, ...) but also fine-grained (e.g. different dog breeds).

The data is split into a *training* and a *validation* set. We train the models on the training set with about 1.28 million images (732 to 1300 per class). Evaluation happens on the validation set of 50000 images (50 per class).

4.2 Limitations

In this section we describe the two main limitations of our implementation, and how they affect our evaluation setup.

4.2.1 Distribution Awareness

Unlike other `tf.Optimizer` classes, the `EHNewtonOptimizer` does not support the `tf.distribute` API. That API is responsible for coordinating data-parallel computations on multiple devices. As a result, our Optimizer cannot be used to train synchronously on multiple devices (like GPUs). Additionally, our Optimizer has to be explicitly placed on the device that should run its computations. This can be achieved by wrapping the instantiation of the object inside a `tf.device` scope, e.g.:

```
with tf.device('/GPU:0'):
  e = EHNewtonOptimizer(...)
```

4.2.2 Memory Requirements

The main limiting factor of our implementation is its graph memory requirement. Before being executed, the optimization procedure has to be compiled into a TensorFlow graph first (see 3.1). Like every other `tf.Optimizer`, we have to compile the optimization procedure separately for every trained variable in the model graph. Unlike other optimizers, we have to compute (second-order) gradients inside our procedure. To compute these gradients, TensorFlow has to copy graph-information to each procedure. The amount of graph information required scales linearly with the variable depth: output variables don't require additional information, input variables need the entire model graph. This is a technical, not an algorithmic, limitation. Copying the graph structure is not necessary in principle, since all gradients are taken over the same model.

As a consequence, our Optimizer has quadratic memory requirements. Performance is severely impacted if we try to train an entire deep model. For this reason, we restrict ourselves to training the last layer of the convolutional models. Fortunately, this already gives insights into the behaviour and performance of the EHNewton algorithm.

4.3 The Training Procedure

4.3.1 Models and Loss Functions

As mentioned in 3.1.3, we use the modified `train` and `eval` scripts provided by the TF-Slim image classification model library (Slim-CML) to execute the runs. Both run in parallel, with training using the primary Titan XP GPU. For all three models, the Slim-CML implementations are used.

Each model has as output a vector $\boldsymbol{f}$ of $K = 1001$[1] logits. They are fed into the softmax activation function $\sigma(\boldsymbol{f})_i = \frac{e^{\boldsymbol{f}_i}}{\sum_{j=1}^{K} e^{\boldsymbol{f}_j}}$. The softmax outputs serve as inputs to the cross-entropy loss (see 2.1.3).

Additionally, each model will define *regularization* losses. Regularization losses work on just the model parameters, for example by penalizing very large weights or by encouraging weight sparsity [1]. These measures intend to improve the *generalization* capabilities of a model (but not necessarily improve its train error). While it is an active topic of research, covering its theory is outside the scope of this thesis. For an overview, we refer to Chapter 7 *Regularization for Deep Learning* in [1].

The sum of cross-entropy loss and the regularization losses forms the *total loss*, which is then minimized by our optimizers.

4.3.2 Finding Suitable Hyperparameters

For the four first-order algorithms Adam, RMSProp, AdaGrad and SGD we stick mostly to default hyperparameter values, consistent with literature [1] and practice [34]. The following values have been used to train all three models:

- Adam: $\beta_1 = 0.9$ and $\beta_2 = 0.999$ and $\epsilon = 10^{-7}$ (for stability)
- RMSProp: $\rho = 0.9$ and $\epsilon = 10^{-7}$ (for stability)

[1]The ResNet-50 implementation only supports 1000 classes – dropping the 1001th, *empty*, class [34].

Optimizer	Start LR	Decay Rate	Decay Every _ Epochs
EHNewton	0.001	0.94	0.05
Adam	0.0001	0.94	0.05
RMSProp	0.0001	0.94	0.05
AdaGrad	0.001	0.94	0.05
SGD	0.0001	0.94	0.1

Table 4.1: Learning Rates and Decay Schedules for Training InceptionV3

- SGD: Plain SGD was not able to make meaningful progress. Adding *momentum* [14] of 0.9 greatly improved its performance.

To find a good learning rate, we use a rather simple heuristic. First, we sweep over the set $\left\{10^{-i} : i \in [7]\right\}$. For the i offering the greatest progress (lowest loss) after 300 steps, we also try increasing values from $\left\{n * 10^{-i} : n \in [9]\right\}$ until one does not offer further improvement. The chosen learning rate is decayed exponentially. Decay rate and -speed vary between models and algorithms.

4.4 Results

4.4.1 InceptionV3

We train the last layer of InceptionV3. For this, we intialize all layers, except the last, from a pre-trained checkpoint linked to by Slim-CML (achieving top-1 accuracy of 78.0% and top-5 recall of 93.9%).

Inception networks (see 2.4.1) also include auxiliary (aux) logits. For training, we consider both the logit and aux logit layers. Their losses are added with weight 1.0 for the main logits and 0.4 for the aux logits. The Slim-CML implementation of InceptionV3 includes *L2-regularization* (of weight $\lambda = 0.00004$) on all convolutional and fully-connected layers. L2 loss over a weight set $\boldsymbol{W}$ is defined by $\lambda \sum_{w \in \boldsymbol{W}} w^2$. Batchnorm (decay 0.9997, $\epsilon = 0.001$) is applied after all convolutional layers.

For EHNewton, we find that a maximum of 20 CG-iterations and a convergence tolerance of 10^{-5} proved accurate. Further decreasing the tolerance or increasing the number of iterations slows down training without benefiting convergence. A fairly large Tikhonov regularization of $\tau = 0.01$ improves convergence and stability.

Batch size for training is 64, the largest possible multiple-of-8 batch size that fits on the Titan XP GPU. We train for 20100 steps, one full epoch[2] of the ILSVRC set. Table 4.1 shows the used learning rate schedules for each algorithm. Since an epoch is almost exactly 20019 steps (of size 64), decaying every 0.05 epochs translates to 1000 steps.

Consistent with literature [2], the second-order algorithm supported a larger learning-rate, as it scales its steps using more accurate curvature information. Also to note, AdaGrad needs a $10\times$ larger LR than RMSProp to counteract its excessive early slowdown.

[2]We save checkpoints every 300 steps. 20100 is the smallest multiple of 300 after one full epoch.

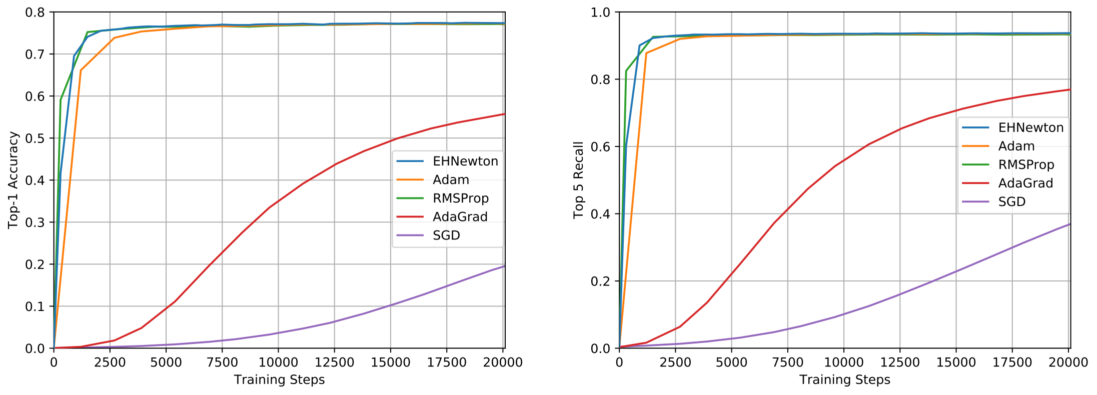


Figure 4.1: Inception Top-1 Accuracy (left) and Top-5 Recall (right) over Training Steps.

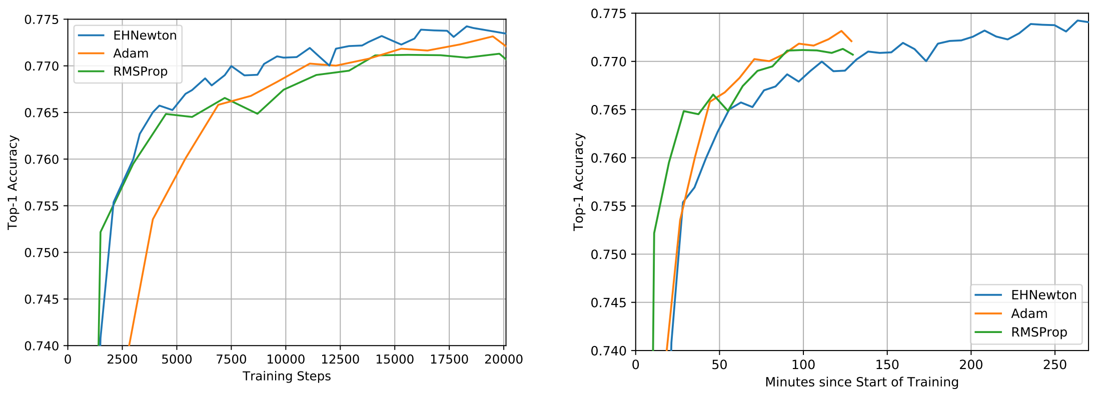
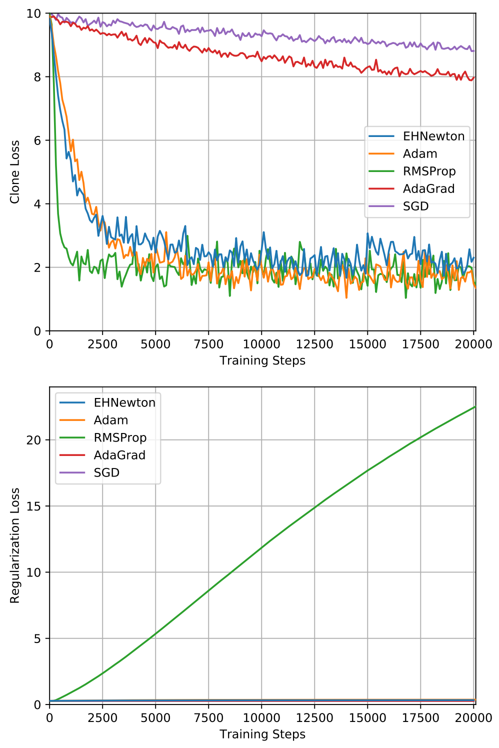


Figure 4.2: Close-up: Inception Accuracy over Training Steps (left) and Time (right).

Figure 4.1 shows the top-1 accuracy ($\frac{\text{\#correct predictions}}{\text{\#of predictions}}$) and the top-5 recall (no. of predictions containing the correct result in the top-5 outputs divided by the no. of predictions) over the number of train steps. Both figures show SGD and even AdaGrad clearly outclassed by EHNewton, Adam and RMSProp. The latter three show similar performance: rapid improvement at first, followed by only tiny incremental changes.

To highlight the differences between EHNewton, Adam and RMSProp, Figure 4.2 shows a close-up of accuracy over both steps and elapsed training time. We can see EHNewton consistently outperforming the other two on per-step values after ca. 3000 steps. It finishes at almost 77.4%, very close to the pre-trained value of 78.0%. Adam comes second with a final accuracy of 77.2% followed by RMSProp with 77.1%. The per-step performance comes at a higher computational cost. This is reflected in the per-time performance, where both Adam and RMSProp reach higher accuracies much faster. In fact, it takes EHNewton twice as long before its accuracy first overtakes Adam's highest value. Top-5 recall behaves similarly: EHNewton tops out at 93.7%, just shy of the pre-trained 93.9%, followed by Adam (93.6%) and RMSProp (93.3%).



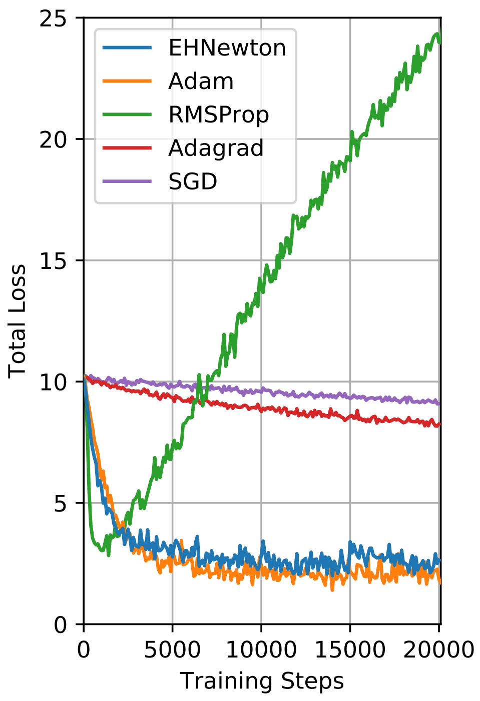


Figure 4.3: Inception Clone Loss, Regularization Loss (left) and Total Loss (right).

The algorithms achieve their accuracy in quite different ways. Figure 4.3 shows the total loss and its components over the number of steps. The clone loss is the weighted sum of cross-entropy for logits and aux. logits. Regularization loss surges when training with RMSProp, whereas it remains almost constant for all other algorithms. Adam converges to the lowest cross-entropy and total loss values. It remains consistently below EHNewton's, even though the latter produces higher accuracies. The up-to-$5\times$ higher loss values of AdaGrad are also surprising, given its 55.7% final accuracy.

4.4.2 ResNet-50

To train the last layer of ResNet-50, a 50-layer variant of ResNet, we again use a pre-trained checkpoint linked in [34]. It achieves a top-1 accuracy of 75.2% and top-5 recall of 92.2%. All layers, except the last layer, are initialized from this checkpoint.

ResNet has just a single final layer of logits. Apart from its cross-entropy loss, we also have regularization losses. For all convolutional layers, an L2 loss is defined, with weight $\lambda = 0.0001$.

For EHNewton we find that, remarkably, performing only a single CG-iteration per step leads to as-good-or-better per-step performance than doing 20 CG-iterations. Since this approach is much cheaper computationally, we show the results of single-iteration EHNewton compared to the first-order optimizers. Tikhonov regularization is $\tau = 0.01$.

Optimizer	Start LR	Decay Rate	Decay Every _ Epochs
EHNewton	0.001	0.94	0.05
Adam	0.0006	0.94	0.05
RMSProp	0.0001	0.94	0.05
AdaGrad	0.002	0.94	0.1
SGD	0.0001	0.94	0.1

Table 4.2: Learning Rates and Decay Schedules for Training ResNet-50

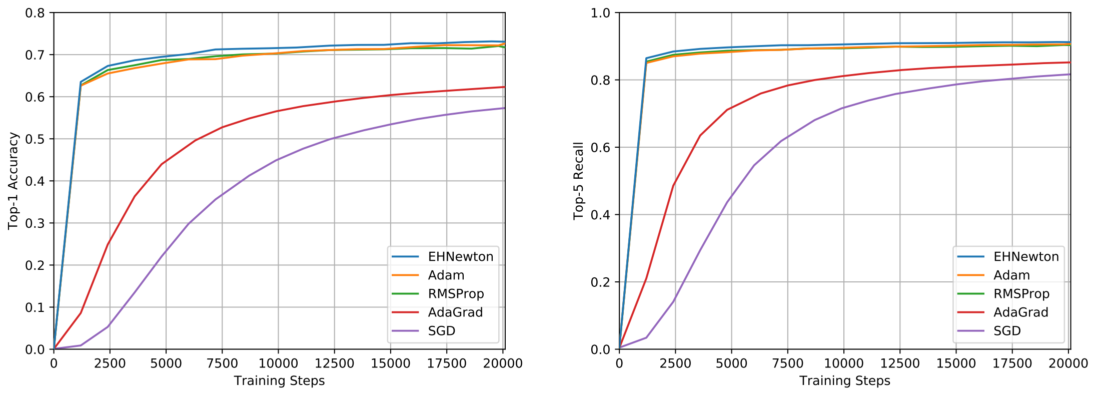


Figure 4.4: ResNet Top-1 Accuracy (left) and Top-5 Recall (right) over Training Steps.

Batch size for training is again 64 and we train for 20100 steps, one full epoch of the ILSVRC set. Table 4.2 shows the learning rate schedules for the algorithms. Like for Inception training, the second-order algorithm supports a higher learning rate than most first-order methods. This time AdaGrad needs an even higher LR (20× RMSProp's) and slower decay to counteract its slowdown.

Figure 4.4 shows the top-1 accuracy and the top-5 recall over the number of train steps. Again, SGD and AdaGrad are outclassed by EHNewton, Adam and RMSProp. This time, however, the former two get final accuracy and recall values much closer to the better performing algorithms. EHNewton more decisively outperforms Adam and RMSProp, posting higher accuracy and recall values for every step after 1200.

Nonetheless, their values are still very close and can be better viewed more close-up in Figure 4.5. We can see a consistent 1% to 1.5% margin between EHNewton per-step accuracy values and those of Adam and RMSProp. Since we only perform one CG-iteration, our algorithm didn't take any longer for training than Adam or RMSProp. Consequently, EHNewton reaches higher accuracies in fewer steps *and* in less wall time.

The final accuracy values: EHNewton reaches 73.1%, followed by Adam with 72.6%, RMSProp with 71.7%, AdaGrad with 62.3% and finally SGD with 57.3%. Top-5 recall behaves similarly to the accuracy: EHNewton achieves 91.2% and Adam gets 90.1%, *below* RMSProp at 90.3%. AdaGrad is not far behind with 85.2%, followed by SGD at 81.6%.

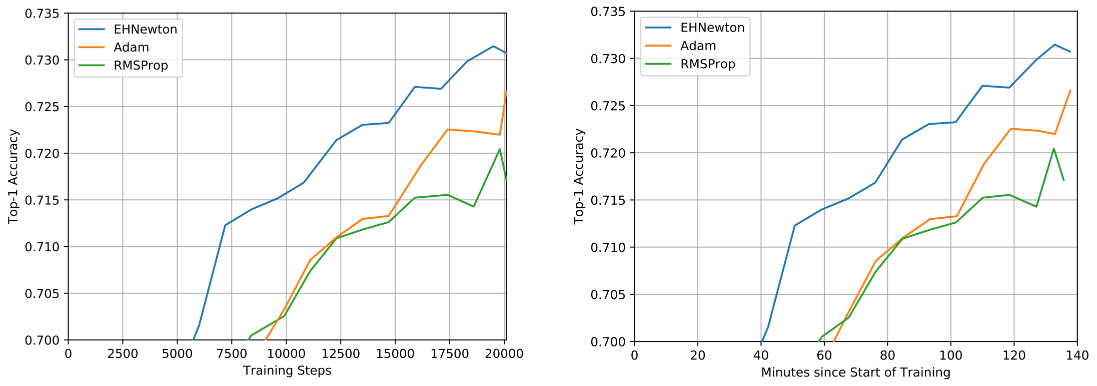


Figure 4.5: Close-up: ResNet Accuracy over Training Steps (left) and Time (right).

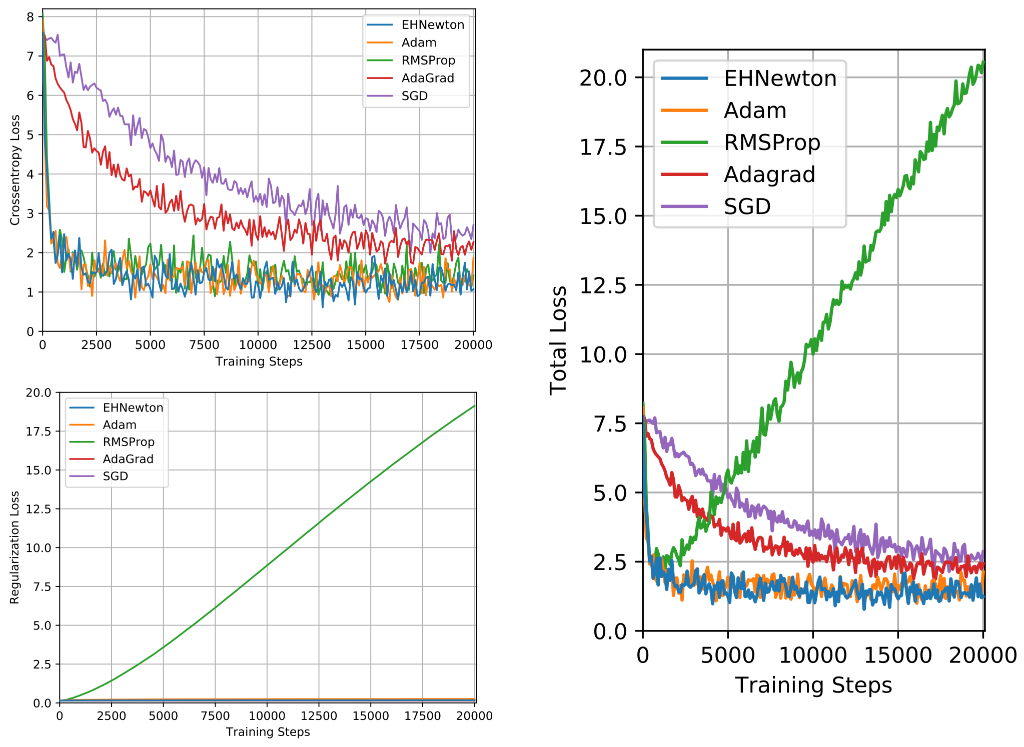


Figure 4.6: ResNet Cross-Entropy and Regularization Loss (left) and Total Loss (right).

Optimizer	Start LR	Decay Rate	Decay Every _ Epochs
EHNewton	0.005	0.90	0.05
Adam	0.0006	0.94	0.05
RMSProp	0.0001	0.95	0.05
AdaGrad	0.002	0.94	0.1
SGD	0.001	0.94	0.05

Table 4.3: Learning Rates and Decay Schedules for Training MobileNetV2

Figure 4.6 shows the total loss and its components, cross-entropy and regularization, over the number of steps. Like during the Inception training, RMSProp increases regularization loss drastically, whereas the other algorithms leave it roughly constant. Adam, EHNewton and RMSProp all converge to similar cross-entropy values (the lowest points reached by EHNewton). AdaGrad and SGD both decrease their loss much further than during Inception training.

4.4.3 MobileNetV2

We train the last layer of MobileNetV2. Its width-multiplier is $\alpha = 1.4$ and input images have a resolution of $\rho = 224 \times 224$. All layers, except the last, are initialized from a checkpoint linked in [34]. MobileNet with all checkpoint parameters achieves a top-1 accuracy of 74.9% and top-5 recall of 92.5%.

MobileNet has a single final layer of logits. Its cross-entropy loss is accompanied by L2-regularization on all (non-separable) convolutional layers, with weight $\lambda = 0.00004$.

EHNewton, like during ResNet training, shows similar performance with 1 CG-iteration per step as it does with 20 such CG-iterations. Again, Tikhonov regularization of $\tau = 0.01$ improves convergence and stability.

As MobileNet is smaller than ResNet or Inception, we can fit a batch size of 96 on the Titan XP GPU. Accounting for the larger batch size, we take only 15300 steps. A little more than one full epoch of the ILSVRC set, this matches the training times of the previous two models.

Table 4.3 shows the learning rate schedules for each algorithm. As a full epoch is almost 13345 steps, decaying every 0.05 epochs translates to 667 steps. The second-order optimizer still supports the highest LR, but needs a stronger decay for stability towards the end of training. AdaGrad once again needs a high learning rate, but even SGD profits from an increased rate.

Figure 4.7 shows the top-1 accuracy and the top-5 recall over the number of train steps. AdaGrad and SGD are again outclassed by EHNewton, Adam and RMSProp. But SGD performs better than AdaGrad, reaching an almost 20% higher accuracy and top-5 recall. EHNewton, Adam and RMSProp show rapid improvement at first, but converge to very similar accuracy values.

The differences between these three algorithms is highlighted in Figure 4.8. It shows a close-up of their accuracy values over train steps and elapsed time. RMSProp and

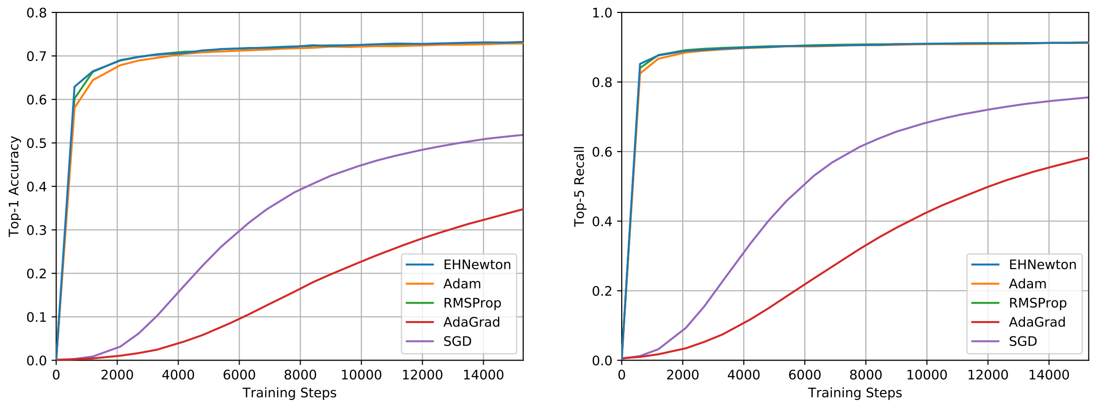


Figure 4.7: MobileNet Top-1 Accuracy (left) and Top-5 Recall (right) over Training Steps.

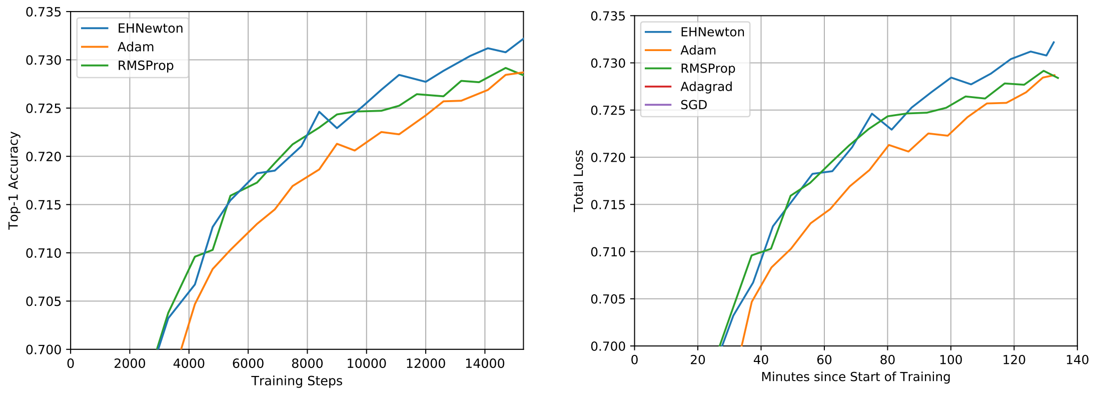


Figure 4.8: Close-up: MobileNet Accuracy over Training Steps (left) and Time (right).

EHNewton produce very similar per-step accuracy values, until EHNewton overtakes at around step 10000. Both have slightly higher values than Adam, although Adam catches up to RMSProp in the last 300 steps. EHNewton finishes at 73.2% accuracy, ahead of Adam with 72.9% and RMSProp at 72.8%. Since we only perform a single CG-iteration, our algorithm didn't take any longer for training than Adam or RMSProp. Thus, again, EHNewton reaches the highest accuracy in fewer steps *and* in less wall time. Top-5 recall behaves similarly: EHNewton tops out at 91.4% and Adam achieves 91.2%, *below* RMSProp at 91.4%.

Figure 4.9 shows the total loss and its components, cross-entropy and regularization, over the number of train steps. Once again, regularization loss surges when training with RMSProp, but remains almost constant for all other algorithms. EHNewton, Adam and RMSProp quickly converge to similarly low cross-entropy values. This time, SGD decreases its loss faster than AdaGrad, but both don't reach loss values as low as during ResNet training.

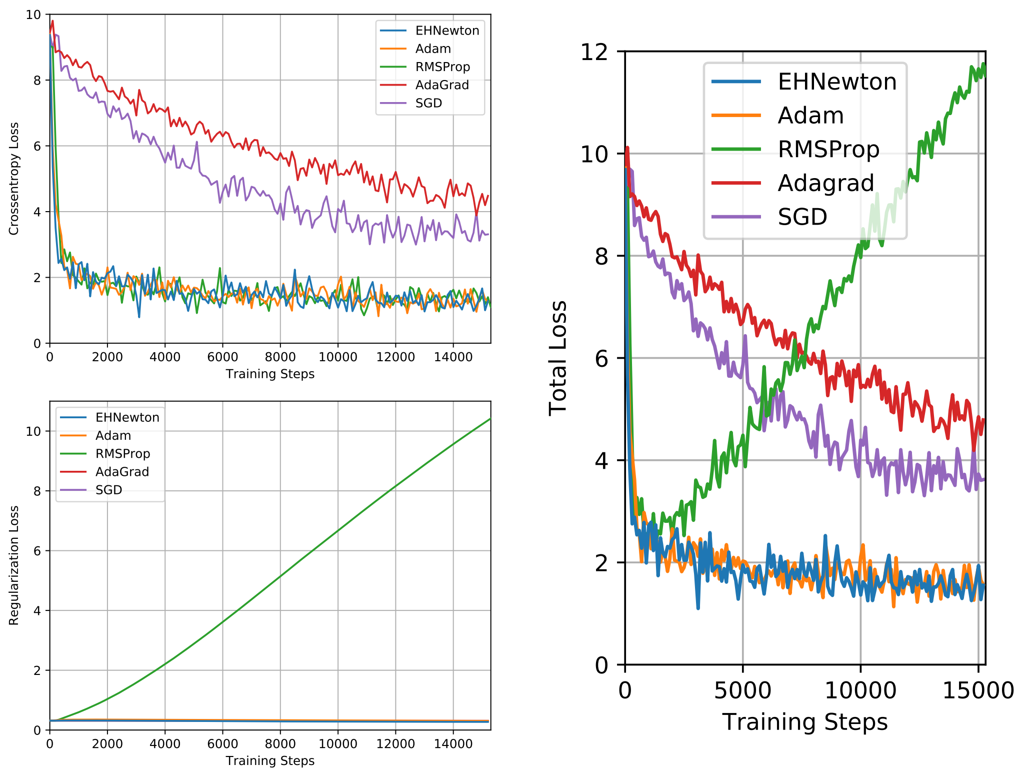


Figure 4.9: MobileNet Cross-Entropy and Regularization Loss (left) and Total Loss (right).

5 Conclusion and Outlook

In this thesis we gave an overview of Neural Network training and its challenges. The most common first-order algorithms were presented, before we introduced the new second-order optimizer EHNewton.

We then implemented the new algorithm in TensorFlow, so that it can be used to train any model defined as a graph in TF. This also made our implementation compatible with the higher-level library TF-Slim.

To compare EHNewton to the first-order optimizers Adam, RMSProp, AdaGrad and SGD, we trained the last layer of three CNNs on an ImageNet dataset. EHNewton showed higher per-step top-1 accuracy and top-5 recall for the models InceptionV3, ResNet-50 and MobileNetV2. A more exact approximation of the solution to the newton equation via 20 CG-iterations led to greater performance on InceptionV3, but at the cost of much increased computational times. Surprisingly, only performing one CG-iteration didn't negatively impact per-step performance when training ResNet-50 and MobileNetV2. In these cases EHNewton reached higher accuracies in fewer steps *and* in less time.

Possible future work could include a thorough investigation of its hyperparameters, especially regarding the surprisingly good performance of its single-CG-iteration variant.

Also, the technical limitations of our implementation have to be addressed, before the algorithm can be used in practice. Especially the "quadratic" memory requirement prevented us from running experiments on more than a few layers at a time. Once this problem is solved, EHNewton can prove to be a practical alternative to first-order algorithms.

In conclusion, despite some limitations of the implementation, EHNewton proved an effective and efficient optimization algorithm. This shows that even simple second-order approaches can still be powerful.

List of Figures

List of Tables

Bibliography

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[2] J. Martens, “Second-order optimization for neural networks,” Ph.D. dissertation, University of Toronto, 2016.

[3] J. Suk, “Application of second-order optimisation for large-scale deep learning,” Master’s thesis, Technical University of Munich, Munich, 2020.

[4] J. Schmidhuber, “Deep learning in neural networks: An overview,” *Neural Networks*, vol. 61, pp. 85 – 117, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608014002135

[5] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[6] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. [Online]. Available: https://doi.org/10.1038/323533a0

[7] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

[8] J. Nocedal and S. Wright, *Numerical optimization*. Springer Science & Business Media, 2006.

[9] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, “Automatic differentiation in machine learning: a survey,” *Journal of Marchine Learning Research*, vol. 18, pp. 1–43, 2018.

[10] G. E. Hinton and R. R. Salakhutdinov, “Reducing the dimensionality of data with neural networks,” *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[11] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[12] H. Robbins and S. Monro, “A stochastic approximation method,” *The Annals of Mathematical Statistics*, vol. 22, pp. 400–407, 1951. [Online]. Available: https://doi.org/10.1214/aoms/1177729586

[13] L. N. Smith, "Cyclical learning rates for training neural networks," in *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 2017, pp. 464–472.

[14] B. Polyak, "Some methods of speeding up the convergence of iteration methods," *Ussr Computational Mathematics and Mathematical Physics*, vol. 4, pp. 1–17, 12 1964.

[15] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International Conference on Machine Learning*, 2013, pp. 1139–1147.

[16] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 2011.

[17] T. Tieleman and G. Hinton, *Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude.* COURSERA: Neural Networks for Machine Learning, 2012.

[18] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[19] A. N. Tikhonov and V. Y. Arsenin, "Solutions of ill-posed problems," *Wiley*, pp. 1–30, 1977.

[20] B. A. Pearlmutter, "Fast exact multiplication by the hessian," *Neural computation*, vol. 6, no. 1, pp. 147–160, 1994.

[21] R. S. Dembo and T. Steihaug, "Truncated-newton algorithms for large-scale unconstrained optimization," *Mathematical Programming*, vol. 26, no. 2, pp. 190–212, 1983.

[22] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[24] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2818–2826.

[25] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, ser. AAAI'17. AAAI Press, 2017, p. 4278–4284.

[26] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, p. 448–456.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[28] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017. [Online]. Available: https://arxiv.org/abs/1704.04861

[29] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," *arXiv preprint arXiv:1801.04381*, 2018. [Online]. Available: https://arxiv.org/abs/1801.04381

[30] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam, "Searching for mobilenetv3," *arXiv preprint arXiv:1905.02244*, 2019. [Online]. Available: https://arxiv.org/abs/1905.02244

[31] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[32] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[33] S. Guadarrama and N. Silberman, "TensorFlow-Slim: A lightweight library for defining, training and evaluating complex models in tensorflow," 2016, [accessed 17-June-2020]. [Online]. Available: https://github.com/google-research/tf-slim

[34] ——, "Tensorflow-slim imgage classification model library," 2016, [accessed 17-June-2020]. [Online]. Available: https://github.com/tensorflow/models/tree/master/research/slim

[35] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.