# Efficient Hessian-Free Optimization of Deep Neural Networks

Niklas Brunn
*Albert Ludwigs University of Freiburg*
*Mathematical Institute*
Freiburg, Germany
niklasbrunn@web.de

Noël E. Kury
*Albert Ludwigs University of Freiburg*
*Mathematical Institute*
Freiburg, Germany
nekury@wkury.de

Clemens A. Schächter
*Albert Ludwigs University of Freiburg*
*Mathematical Institute*
Freiburg, Germany
clemens.schaechter@live.com

***Abstract*—This report is a written elaboration of a project which was developed in the lecture Numerical Optimization in the winter term 21/22 at the Albert Ludwigs University of Freiburg. We discuss a 2nd-order optimization method for training deep neural networks using the generalized Gauss-Newton matrix as an approximation for the Hessian of the objective loss function. In addition we present an implementation of the method using Python3, Tensorflow and compare the results to a 1st-order optimization method on the MNIST database.**

## I. Introduction

Deep neural networks (DNNs) are an important deep learning architecture and have a wide field of applications. When training a DNN, the network parameters are updated according to certain rules, which we call the DNN optimization method. Stochastic gradient descent (SGD) is often used as such an optimization method. SGD is a 1st-order optimization method in which the negative gradient of the objective loss function (OL) with respect to the network parameters is used to update the current parameters. One problem of the SGD method is that it does not consider the pathological curvature of the OL. This means that the method can get stuck in saddle points, or may converge slowly in environments with pathological curvature.

On the other hand 2nd-order optimization methods take the curvature of the OL into account and can thus circumvent such problems. However they are often costly in time because 2nd-order derivatives must be computed for this purpose.

Our aim is to use a version of Newton's method for DNN optimization as presented in [4]. We use an approximation of the Hessian matrix, which is positive definite, and thus allows the use of a preconditioned version of the conjugate gradient (PCG) method for updating the network parameters.

We shortly discuss the individual problems of standard Newton updates and the improvements proposed for them in order to ensure computational efficiency. Our own implementation can be accessed on GitHub .

Finally we train a DNN on the MNIST database with SGD and the presented 2nd-order optimization method and discuss our results.

## II. Optimization of Deep Neural Networks

In this section, we introduce the necessary mathematical notations for optimizing DNNs, formulate the parameter optimization task as a nonlinear programming problem (NLP) and define the concept of matching loss functions.

### A. Deep Neural Networks

A deep neural network is an artificial neural network in which layers between input and output layer provide some depth. Given a data set consisting of observation pairs $D = \{(x_i, y_i)\}_{1\leq i\leq N}$ with inputs $x_i$ and corresponding targets $y_i$, we aim to find a set of parameters such that the DNN's evaluations are approximately the corresponding observed targets $y_i$. In contrast to convention we define the realization of a DNN given a fixed input $x$ as

$$\begin{aligned} R_x &: \Theta^d \to \mathbb{R}^m, \\ \theta &\mapsto R_x(\theta) =: z_x. \end{aligned} \tag{1}$$

Here $\theta$ is a vector containing the parameters of our DNN and $z_x$ is defined as the DNN's output vector. The parameter space is denoted by $\Theta^d = \mathbb{R}^d$ with usually $d >> 0$. More precisely the realization is an alternating composition of a fixed number of affine and nonlinear mappings with respect to its input $x$. The parameter vector is the collection of every entry from all weight matrices and bias vectors in the affine mappings. These affine mappings are called layers and the nonlinear mappings are called activation functions. Activation functions $\varrho : \mathbb{R} \to \mathbb{R}$ are one-dimensional, nonlinear, real-valued and piecewise differentiable functions which are applied element-wise to the activation of the preceding layer. Usually the same activation function is applied to all hidden layers of the DNN.

We assume that a nonlinear function $\phi : \mathbb{R}^m \to \mathbb{R}^m$ is applied to the DNN's output which is not directly considered as a component of the DNN's architecture. This allows us to use technical tricks that later favor the implementation. We denote the nonlinear evaluation of the network output by

$$\hat{y}_x := \phi(z_x), \tag{2}$$

which serves as an approximation of the target $y$. For cases in which no further mapping of the network output is needed, we also allow $\phi$ to be the identity function.

### *B. Loss Functions*

Choosing an appropriate Loss function

$$\begin{aligned} L &: \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^+, \\ (\hat{y}, y) &\mapsto L(\hat{y}, y). \end{aligned} \tag{3}$$

is an important task in parameter optimization of DNNs. The Loss function gives feedback about how well the DNN fits the observation data set. Further we define the output loss function given a fixed target $y$ as

$$\begin{aligned} L_y &: \mathbb{R}^m \to \mathbb{R}^+, \\ \hat{y} &\mapsto L_y(\hat{y}) := L(\hat{y}, y), \end{aligned} \tag{4}$$

which is assumed to be at least twice continuous differentiable in $\hat{y}$. For a fixed observation pair $(x, y)$ with current parameters $\theta$, the output loss takes $\hat{y}_x = \phi(R_x(\theta))$ as input. A typical choice for the loss function $L$ and the previous nonlinear function $\phi$ in multi-classification tasks is cross-entropy loss with a preceding softmax function and squared error loss with identity function for regression tasks.

### *C. Network Training Formulated as a NLP*

Parameter optimization of a DNN can be formulated as an unconstrained NLP. Therefore we define the OL as a mapping which calculates the empirical risk of the DNN‘s output with current parameters $\theta$ given the observation data set

$$\begin{aligned} f_D &: \Theta^d \to \mathbb{R}^+, \\ \theta &\mapsto f_D(\theta) := \mathbb{E}_D[L_y(\hat{y}_x)] = \frac{1}{N}\sum_{j=1}^{N} L_{y_j}(\hat{y}_{x_j}). \end{aligned} \tag{5}$$

We get the unconstrained NLP formulation

$$\underset{\theta \in \Theta^d}{\arg\min} \quad f_D(\theta), \tag{6}$$

where $\theta$ denotes the decision variable and $f_D$ the objective function of the NLP. Further, if we consider only one observation $(x, y)$ from the observation data set $D$, we simply write $f$ for the corresponding OL. Note that the above formulated NLP is typically non convex due to the non convexity of the OL.

### *D. Matching Loss Functions*

For later purposes we need to calculate the Jacobian or the gradient of the OL with respect to the DNN‘s parameters. For simplicity we consider the OL for only one pair of observations

$$J_f := \frac{\partial}{\partial \theta} f(\theta) \tag{7}$$

and neglect the dependency to the parameters $\theta$ in notation. Due to linearity of the derivative, the results can be extended without problems to the case where the whole data set is being considered. Applying the chain rule we can rewrite the Jacobian in (7) to

$$J_f = J_{L_y \circ \phi \circ R_x} = J_{L_y} J_\phi J_{R_x}, \tag{8}$$

$$J_f^{\mathrm{T}} = J_{R_x}^{\mathrm{T}} J_\phi^{\mathrm{T}} J_{L_y}^{\mathrm{T}}. \tag{9}$$

As introduced in [8], we also make use of so-called matching loss functions. An output loss function $L_y$ matches the output nonlinearity $\phi$ iff

$$\left(\frac{\partial}{\partial \hat{y}_x}\left(L_y \circ \phi\right)(\hat{y}_x)\right)^{\mathrm{T}} = J_{L_y \circ \phi}^{\mathrm{T}} = A\hat{y}_x + b, \tag{10}$$

for a matrix $A$ and a vector $b$, which both do not depend on the network parameters $\theta$.
Note that cross-entropy loss matches the softmax function and squared error loss matches the identity function.

## III. Hessian-Free Optimization Method

In this section we want to provide the reader a compact summary of the different components used in the presented optimization method. We will briefly review Newton‘s method and discuss the disadvantages of using the Hessian matrix directly. We then introduce the necessary theory for the efficient 2nd-order optimization method called Hessian-free method. For a more detailed introduction in Newton-type optimization we refer the reader to chapter 7 in [2]. For details about problems with pathological curvature to [4], and for information about the generalized Gauss-Newton matrix to chapter 8 in [5].
A pseudocode for the full Hessian-free algorithm is given in Algorithm 1 (note that we use the positive gradient in the CG method and thus have to adapt the algorithm).

**Algorithm 1:** Hessian-free pseudocode for (6)

```
Input: $D$, $\theta_0$, $\lambda$, epochs
$\Delta\theta_0 \leftarrow 0$
$k, e \leftarrow 0$
while $e <$ epochs do
    shuffle $D$
    for Batch $B$ in $D$ do
        compute $J_{f_B}^{\mathrm{T}}(\theta_k)$ with BAD
        $\Delta\theta_{k+1}$, denom $\leftarrow$ PCG($B$, $\Delta\theta_k$, $J_{f_B}^{\mathrm{T}}(\theta_k)$, $\lambda$)
        $\theta_{k+1} \leftarrow \theta_k - \Delta\theta_{k+1}$
        $\lambda \leftarrow \lambda$-update($B$, $\Delta\theta_{k+1}$, $\lambda$, 2denom)
        $k \leftarrow k + 1$
    end
    $e \leftarrow e + 1$
end
Output: $\theta^\star$
```

### *A. Newton's Method*

Newton‘s method is an optimization method, where we update the decision variables of our NLP formulation in (6) iteratively by

$$\theta_{k+1} = \theta_k - H_f(\theta_k)^{-1} J_f^{\mathrm{T}}(\theta_k), \tag{11}$$

where $H_f(\theta_k)$ denotes the Hessian of the OL with respect to the current network parameters $\theta_k$. One can obtain this by minimizing the local quadratic approximation

$$\Delta\theta_{k+1} := \theta_{k+1} - \theta_k, \tag{12}$$

$$\begin{aligned} q_{\theta_k}(\Delta\theta_{k+1}) &:= f(\theta_k) + J_f^{\mathrm{T}}(\theta_k)\Delta\theta_{k+1} + \\ &\quad \frac{1}{2}\Delta\theta_{k+1}^{\mathrm{T}} H_f(\theta_k)\Delta\theta_{k+1} \qquad (13) \\ &\approx f(\theta_k + \Delta\theta_{k+1}), \qquad (14) \end{aligned}$$

with respect to $\Delta\theta_{k+1}$.
Newton's method can converge in fewer iterations than gradient descent. This is because it rescales the gradient of the OL in every step by using information about the local curvature of the OL. This makes Newton's method less susceptible to pathological curvature, than 1st-order gradient based methods like SGD. We will also neglect the dependency on the parameters $\theta$ in our notation of the Hessian and Hessian approximations, whenever the matrices are not mentioned in the context of a Newton step.

## B. Problems with the Hessian

We assume in every step (11) that the Hessian is invertible, which sometimes is not the case in our task (6). Since the Hessian is not necessarily positive semidefinite either, convergence to a local minimum can not be ensured. Furthermore the Hessian has $d^2$ entries consisting of 2nd-order partial derivatives, which makes it expensive to calculate, assuming $d >> 0$. For these reasons, the direct application of Newton's method to our task is inappropriate and should be modified to safeguard an efficient application.

## C. The Generalized Gauss-Newton Matrix

We no longer use the Hessian itself, but the generalized Gauss-Newton matrix (GGN) as an approximation of the Hessian

$$G_f := J_{R_x}^{\mathrm{T}} H_{L_y \circ \phi} J_{R_x}, \tag{15}$$

where $H_{L_y \circ \phi}$ denotes the Hessian of $L_y \circ \phi$ with respect to $z_x$. The matrix $J_{R_x}$ denotes the Jacobian of the realization of the DNN with respect to its parameters. Regarding the full Hessian of the OL

$$H_f = J_{R_x}^{\mathrm{T}} H_{L_y \circ \phi} J_{R_x} \sum_{j=1}^{m} \left( \left( J_{L_y \circ \phi}^{\mathrm{T}} \right)_j H_{(R_x)_j} \right), \tag{16}$$

where $H_{(R_x)_j}$ denotes the Hessian of the DNN's $j$-th output, the last sum-term in (16) vanishes if one of the two factors in the sum vanishes. This means that in a neighborhood of a local optimum $\theta^\star$ it holds that $H_f \approx G_f$. Moreover, by construction, the GGN is symmetric and positive semidefinite if $L_y \circ \phi$ is convex. This is because for every $v \in \mathbb{R}^d$ it holds that

$$\begin{aligned} v^{\mathrm{T}} G_f v &= v^{\mathrm{T}} \left( J_{R_x}^{\mathrm{T}} H_{L_y \circ \phi} J_{R_x} \right) v \qquad (17) \\ &= \left( J_{R_x} v \right)^{\mathrm{T}} H_{L_y \circ \phi} \left( J_{R_x} v \right) \qquad (18) \\ &\geq 0. \qquad (19) \end{aligned}$$

For both cross-entropy loss with previous softmax function and squared error loss with previous identity function it holds that $L_y \circ \phi$ is convex.

## D. Levenberg-Marquardt Algorithm

To ensure invertibility we use a damped version of the GGN (DGGN). The idea is to use a Levenberg-Marquardt algorithm, where we add a diagonal matrix with only positive values, i.e.

$$DG_f := G_f + \lambda \cdot I_d, \tag{20}$$

with $I_d$ denoting the $d \times d$ unit matrix and some $\lambda > 0$. The DGGN is positive definite. To see this, we consider the decomposition

$$G_f = U \Lambda U^{\mathrm{T}}, \tag{21}$$

with an orthogonal Matrix $U \in \mathbb{R}^{d \times d}$ containing the eigenvectors to the corresponding eigenvalues as columns and a diagonal Matrix $\Lambda \in \mathbb{R}^{d \times d}$ with the non negative eigenvalues of the GGN on its diagonal entries. This decomposition exists because the GGN is symmetric and positive semidefinite. Further it holds that

$$\begin{aligned} DG_f &= U \Lambda U^{\mathrm{T}} + \lambda \cdot I_d \qquad (22) \\ &= U \Lambda U^{\mathrm{T}} + \lambda \cdot U I_d U^{\mathrm{T}} \qquad (23) \\ &= U \left( \Lambda + \lambda \cdot I_d \right) U^{\mathrm{T}}, \qquad (24) \end{aligned}$$

where the diagonal Matrix $(\Lambda + \lambda \cdot I_d)$ only contains positive entries, the eigenvalues of the DGGN. It follows that this matrix is positive definite and invertible.
Note that a choice of $\lambda >> 0$ leads to a Newton step where the change of our parameters is marginal. On the other hand $\lambda \approx 0$ leads to a full step using the GGN. In practice, we update the damping value $\lambda$ after each iteration. For this reason, the damping parameter can be seen as a self-adapting learning rate for the optimization method.
Following [4] we define the reduction ratio

$$\rho := \frac{f(\theta_k) - f(\theta_{k+1})}{q_{\theta_k}(\Delta\theta_{k+1}) - q_{\theta_k}(0)} \tag{25}$$

and then update the value $\lambda$ according to the rule

```
if $\rho < \frac{1}{4}$ :
    $\lambda_{k+1} = r \cdot \lambda_k$
elif $\rho > \frac{3}{4}$ :
    $\lambda_{k+1} = r^{-1} \cdot \lambda_k$
else :
    $\lambda_{k+1} = \lambda_k$
return : $\lambda_{k+1}$,                    (26)
```

where $r = 1.5$. We can compute the denominator of $\rho$ within the PCG-run and thus save some computation time (output denom in Alogrithm 2). Otherwise we need to compute an additional matrix-vector-product.
In our experiments we found that for a smaller batch size (e.g.

$N < 150$) it is important to adjust the update amount of the damping value to a smaller size, e.g. $r = 1.01$, to ensure stable convergence (see Fig. 1.). For a large batch size $r = 1.01$ leads to slower convergence.

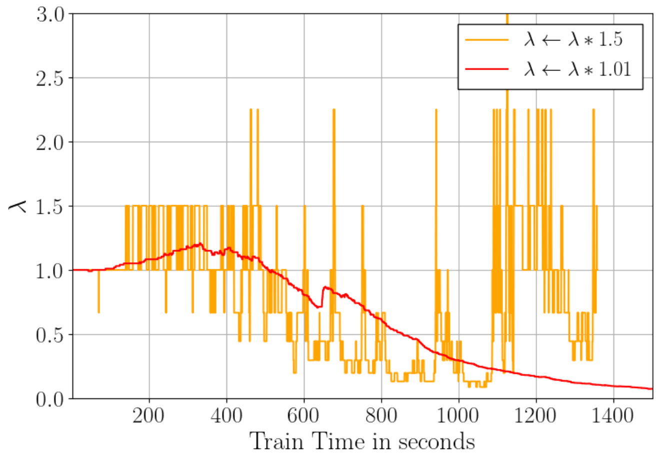

Fig. 1. Damping parameter $\lambda$ during training with batch size 75 on the MNIST database with $r = 1.5$ (orange) and $r = 1.01$ (red). A large $r$-value leads to unstable convergence.

## E. Preconditioned Conjugate Gradient Method

Computing the parameter updates in (11) directly using the DGGN would be costly because we need to calculate and store the inverse of the DGGN. Instead we reformulate the equation in (11) to an equivalent formulation

$$DG_f(\theta_k)\,\Delta\theta_{k+1} = -J_f^{\mathrm{T}}(\theta_k), \tag{27}$$

and approximate the solution to this system of linear equations using a preconditioned version of the CG-method [4].
This approach allows us to efficiently approximate the parameter updates with at most $d$ steps, without calculating and storing the DGGN or its inverse at any time. Instead we use an efficient way to calculate matrix-vector-products for the computation of the DGGN-vector-product within the PCG method.
Since in our task $d >> 0$, we do not iterate until convergence but instead terminate the iterations after a minimum number of steps $t$ if certain termination criteria are met. It is worth mentioning that in our experiments a minimum number of 3 PCG-steps works well even if we have $d = 636,010$ or larger.
In Algorithm 2 we provide a pseudocode for our implementation of the PCG method, where $\varepsilon$ represents an accuracy for the termination criteria and $\langle .\,,\,. \rangle$ denotes the standard scalar product.

## F. Efficient Computation of Matrix-Vector-Products

We use an efficient and fast way to calculate matrix-vector-products in the PCG-iterations. This idea was first presented in [8] using results from [7].
We consider the product of the DGGN with an arbitrary vector $v$ that we want to compute in an efficient way. For this task we make use of FAD and BAD since it is well known (e.g. chapter 10 in [2]) that for a vector valued function $F : \mathbb{R}^n \to \mathbb{R}^m$ we can compute the Jacobian-vector-product at a cost of only

**Algorithm 2:** PCG-method pseudocode for (27)

```
Input: Batch $B$, $\Delta\theta_k$, $J_{f_B}^{\mathrm{T}}(\theta_k)$, $\lambda$
$i, j, s \leftarrow 0,\ t,\ t$
$v_0 \leftarrow \Delta\theta_k$
compute $w_0 = DG_{f_B} \cdot v_0$ with fast-mat.-vec.
$r_0 \leftarrow J_{f_B}^{\mathrm{T}} - w_0$
$M \leftarrow$ preconditioner matrix [4]
$y_0 \leftarrow M^{-1} r_0$
$d_0 \leftarrow y_0$
$\psi_0 \leftarrow \frac{1}{2} \cdot \langle v_0,\ w_0 - 2 \cdot J_{f_B}^{\mathrm{T}} \rangle$
while $i \leq j$ or $s > \varepsilon \cdot j$ or $\psi_i \geq 0$ do
    $j \leftarrow \max\{t,\ \lceil \frac{i}{t} \rceil\}$
    $i \leftarrow i + 1$
    compute $z_i = DG_{f_B} \cdot d_{i-1}$ with fast-mat.-vec.
    $\alpha_i \leftarrow \frac{\langle r_{i-1},\, y_{i-1} \rangle}{\langle d_{i-1},\, z_i \rangle}$
    $w_i \leftarrow w_{i-1} + \alpha_i \cdot z_i$
    $v_i \leftarrow v_{i-1} + \alpha_i \cdot d_{i-1}$
    $r_i \leftarrow r_{i-1} - \alpha_i \cdot z_i$
    $y_i \leftarrow y_{i-1} - M^{-1} \alpha_i \cdot z_i$
    $d_i \leftarrow y_i + d_{i-1} \cdot \frac{\langle r_i,\, y_i \rangle}{\langle r_{i-1},\, y_{i-1} \rangle}$
    $\psi_i \leftarrow \frac{1}{2} \cdot \langle v_i,\ w_i - 2 \cdot J_{f_B}^{\mathrm{T}} \rangle$
    if $i \geq j$ then
        $s \leftarrow 1 - \frac{\psi_{i-j}}{\psi_i}$
    end
end
denom $\leftarrow \psi_i + \langle v_i,\ 2 \cdot J_{f_B}^{\mathrm{T}} - \frac{1}{2} \cdot \lambda \cdot v_i \rangle$
$\Delta\theta_{k+1} \leftarrow v_i$
Output: $\Delta\theta_{k+1}$, denom
```

$$\mathrm{cost}_{\mathrm{FAD}}(J_F v) \leq 2\,\mathrm{cost}(F), \tag{28}$$
$$\mathrm{cost}_{\mathrm{BAD}}(J_F^{\mathrm{T}} v) \leq 3\,\mathrm{cost}(F). \tag{29}$$

For matching loss functions we get from (10)

$$\begin{aligned} H_{L_y \circ \phi} &= \frac{\partial}{\partial z_x} J_{L_y \circ \phi}^{\mathrm{T}} && (30)\\ &= A J_\phi && (31)\\ &= J_\phi^{\mathrm{T}} A^{\mathrm{T}}. && (32) \end{aligned}$$

This allows us to circumvent the direct computation of the Hessian matrix. We want to calculate the following matrix-vector-product

$$\begin{aligned} DG_{f_B} v &= \left(J_{R_x}^{\mathrm{T}} A J_\phi J_{R_x} + \lambda \cdot I_d\right) v && (33)\\ &= J_{R_x}^{\mathrm{T}} A J_{\phi \,\circ\, R_x} v + \lambda \cdot I_d v. && (34) \end{aligned}$$

Here we can compute the first part of the sum in (34) using Tensorflow's pre-implemented functions for FAD and BAD. We first compute the outputs of the DNN given the current parameters and afterwards evaluate the Jacobian-vector-product

$$u := J_{\phi \,\circ\, R_x} v \tag{35}$$

with FAD. This is followed by an left multiplication with the matrix $A$ and computation of the product

$$J_{R_x}^{\mathrm{T}} w \tag{36}$$

using BAD, where $w := Au$. Also note that using (32) instead of (31) we get

$$DG_f v = J_{\phi \circ R_x}^{\mathrm{T}} A J_{R_x} v + \lambda \cdot I_d v. \tag{37}$$

In contrast to (34), more of the differentiation task is put into the backward pass.
Note that for both, cross-entropy loss with previous softmax function and squared error loss with previous identity function, we get $A = I_m$. In these two cases one can simplify equation (31) and (32) to

$$H_{L_y \circ \phi} = J_\phi = J_\phi^{\mathrm{T}}. \tag{38}$$

### *G. Mini-Batches*

If we are facing a large observation data set with $N >> 0$, it is common to split the data set into batches $B \subset D$ containing only a few random samples $M << N$ of the entire data.
In every training epoch we apply the optimization algorithm to each generated batch. The partition of $D$ into smaller batches is called a mini-batch scenario, where $M$ denotes the batch size of the mini-batches. For each mini-batch $B$ the DGGN in (20) becomes

$$DG_{f_B} := \frac{1}{M} \sum_{(x,y) \in B} \left( J_{R_x}^{\mathrm{T}} H_{L_y \circ \phi} J_{R_x} \right) + \lambda \cdot I_d. \tag{39}$$

As mentioned in [3], the batch size should be relatively large to capture enough information about the curvature of the OL and to ensure stability in optimization. But we found a small batch size can prevent overfitting. We will discuss this in our experiment section.

## IV. Implementation

We implement our version of the Hessian-free optimization method with Python v.3.8.3 and Tensorflow v.2.4.0. .
We make use of Tensorflows pre-implemented FAD-command tf.autodiff.ForwardAccumulator and BAD-command tf.GradientTape and GPU support. Our code is available at our GitHub repository . Our code can not only be used for optimizing dense neural networks but also convolutional neural networks or other feedforward artificial neural networks.
We train our models on an Asus F571 Laptop with 8GB RAM, Intel Core i5-9300H and GeForce GTX 1650 4GB VRAM.

## V. Experiments

To test our implementation we used the MNIST database and a simple generated sinus data set.

### *A. Simple Simulated Sinus Data*

We generate 2000 random samples of sinus function values with some noise, $x_i \sim \mathcal{N}(0,1)$, $y_i = \sin(x_i) + 0.1 \cdot \beta_i$ with $\beta_i \sim \mathcal{N}(0,1)$. For the network architecture we use two hidden layers with 15 neurons each and ReLu activation function resulting in a total number of parameters $d = 286$. For the output loss we choose the squared error loss with previous identity function. We set the batch size for SGD and the Hessian-free optimizer to $M_{\mathrm{SGD}} = 100$ and $M_{\mathrm{HF}} = 1000$.
We tested several values for the SGD-learning rate. At the end we could determine that our implementation of the Hessian-free method does not only converge in fewer epochs but also can compete with SGD in computation time (see Fig. 2 for a representative result).

### *B. MNIST*

The MNIST database consists of grayscale images of hand-written digits from 0 to 9, with a resolution of $28 \times 28$ pixels. We flatten each image to a vector of size 784 and divide by 255, which transforms the data to $[0,1]^{784}$. The train data set consists of $60,000$ images. For validation we use the test data set with $10,000$ images.
For our DNN we choose an architecture with one hidden layer with 800 neurons and ReLu activation function. The total number of trainable parameters is $636,010$.
For stochastic gradient descent we choose as learning rate $\eta = 0.1$ and batch size $M_{\mathrm{SGD}} = 100$. We also tested $\eta = 0.01$, $\eta = 0.025$ and $\eta = 0.3$, as well as $M_{\mathrm{SGD}} = 250$, but it achieved worse results.
The SGD benchmark for this model architecture with cross-entropy loss and previous softmax function without data preprocessing, distortion, weight decay etc. is an error rate of $1.6\%$ on the test data set. It can be achieved by initially setting the learning rate to $\eta = 0.005$ and multiplying it by 0.3 every 100 epochs. But convergence in this scenario is very slow.
We set the accuracy $\varepsilon = 0.0005$ and the number of minimal PCG-steps to 3. In this setting the PCG-method terminates mostly after $6-10$ steps. A higher minimal PCG-step number e.g. 10 does not lead to better performance, but significantly more time for one total update step is needed. Here PCG-method meets the termination criteria after $13-17$ steps.
With a large batch size ($M_{\mathrm{HF}} = 1000$) the method converges faster than SGD with learning rate $\eta = 0.1$ and $N = 100$. But generally SGD seems to converge more stable. After some time both methods tend to overfit on the train data. Here the total error on the train data converges to 0, which prevents any improvement gains on the test data set.
With a smaller batch size ($M_{\mathrm{SGD}} = 75$) the method converges much slower than SGD with learning rate $\eta = 0.1$ and also seems to be far more unstable in comparison. But this instability prevents overfitting to the train data as the total error on the train data set never converges to 0. At around 1000 seconds the method passes SGD and at 2000 seconds the method consistently beats the benchmark. After around 3000 seconds the best performance measured is a $1.46\%$ error rate

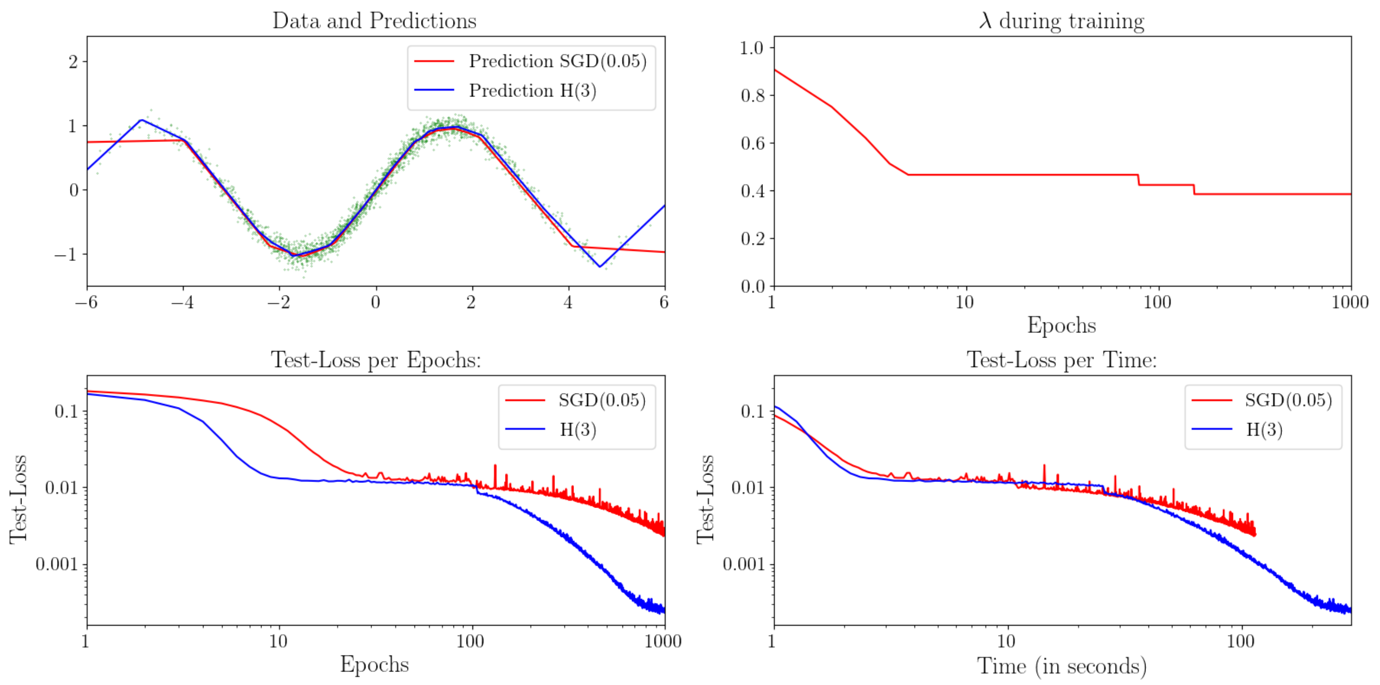

Fig. 2. Comparison of our Hessian-free algorithm (3 CG-steps) with SGD ($\eta = 0.05$). For the $\lambda$-updates we chose $r = 1.01$ for a stable optimization. Even if only a few samples were generated on the borders of the upper left graphic, Hessian-free was able to fit the data more accurate than SGD.

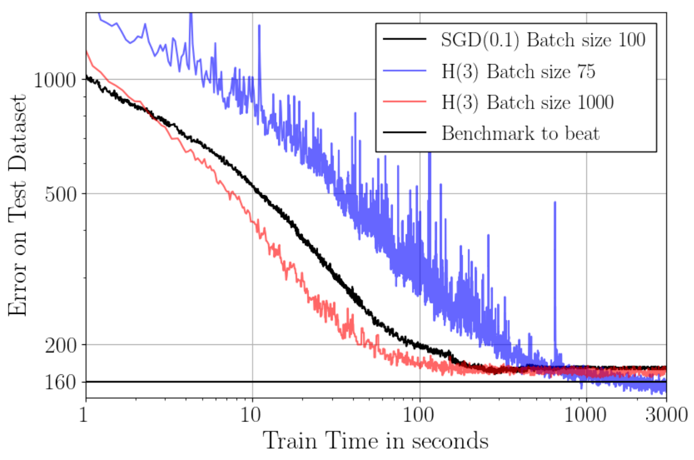

Fig. 3. Comparison with SGD (black) and the effect of the batch size on performance. On the $y$-axis the total number of wrongly classified images on the whole test data set is plotted. The corresponding train time to reach these result is plotted on the $x$-axis.

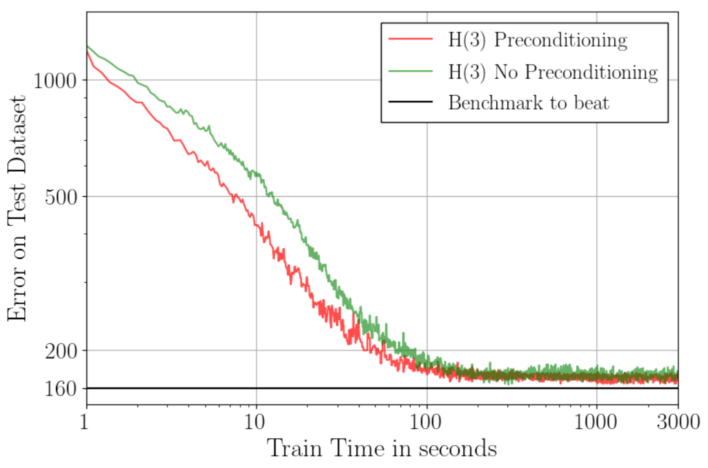

Fig. 4. Comparison of the PCG-method and the CG-method. Without preconditioning convergence is slower. (Tested with $M_{\mathrm{HF}} = 100$)

but due to the stochasticity of mini-batching can get as high as 1.62% after optimizing on some batches. Because minor performance gains are still noticeable it may be possible to achieve an even better error rate. Also the method becomes significantly more stable as the training progresses. Our results are displayed in Fig. 3.
We also tested the effect of using the preconditioned CG-Method rather than the vanilla CG-Method. As can be seen in Fig. 4, preconditioning leads to faster convergence and to slightly better predictions (ca. 0.05%).

## VI. Conclusion

With our implementation of the Hessian-free method we could compete with, and sometimes even beat, SGD in terms of run time and performance. During our work on the project we found it very tricky and time consuming to implement the Hessian-free method.
It would also be interesting to see how our implementation performs with other network architectures or models (e.g. convolutional neural networks, (variational) autoencoders), and possibly without the use of matching-loss functions.
Considering that one can achieve good performances with

modified 1st-order optimization methods (e.g. ADAM) it remains to be seen whether 2nd-order optimization methods can gain a foothold within standard network training methods. On the other hand it could be possible to achieve further improvements in performance with the Hessian-free optimization method and that the potential of 2nd-order optimization methods has not yet been fully uncovered.

## Acknowledgment

We want to thank the lecture assistant Florian Messerer for suggesting this topic for the project and providing us some sources to research.

## References

[1] O. Chapelle, D. Erhan, "Improved Preconditioner for Hessian Free Optimization", Yahoo! Labs

[2] M. Diehl, "Lecture Notes on Numerical Optimization (Preliminary Draft)", Albert Ludwigs University of Freiburg, September 29, 2017

[3] M. Gargiani, A. Zanelli, M. Diehl, F. Hutter, "On the Promise of the Stochastic Generalized Gauss-Newton Method for Training DNNs", arXiv:2006.02409v4, June 9, 2020

[4] J. Martens, "Deep learning via Hessian-free optimization", University of Toronto, Ontario, M5S 1A1, Canada, 2010

[5] J. Martens, "New Insights and Perspectives on the Natural Gradient Method", Jurnal of Machine Learning Research 21, arXiv:1412.1193v11, September 19, 2020

[6] J. Martens, I. Sutskever, "Training Deep and Recurrent Networks with Hessian-Free Optimization", In: G. Montavon, G.B. Orr, KR. Müller (eds), Neural Networks: Tricks of the Trade. Lecture Notes in Computer Science, vol 7700. Springer, Berlin, Heidelberg, 2012

[7] B. A. Pearlmutta, "Fast Exact Multiplication by the Hessian", Neural Computation, June 9, 1993

[8] N. N. Schraudolph, "Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent", Neural Computation, August 2002