



Computational Science and Engineering
(Int. Master's Program)

Technische Universität München

Master's thesis in Computational Science and Engineering

**Preconditioning for Hessian-Free
Optimization**

Robert Seidl





Computational Science and Engineering (Int. Master's Program)

Technische Universität München

Master's thesis in Computational Science and Engineering

Preconditioning for Hessian-Free Optimization

Author: Robert Seidl
1st examiner: Prof. Dr. Thomas Huckle
2nd examiner: Prof. Dr. Michael Bader
Thesis handed in on: April 2, 2012



I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

April 2, 2012

Robert Seidl

Abstract

Recently Martens adapted the Hessian-free optimization method for the training of deep neural networks. One key aspect of this approach is that the Hessian is never computed explicitly, instead the Conjugate Gradient(CG) Algorithm is used to compute the new search direction by applying only matrix-vector products of the Hessian with arbitrary vectors. This can be done efficiently using a variant of the backpropagation algorithm. Recent algorithms use diagonal preconditioners to reduce the needed iterations of the CG algorithm. They are used because of their easy calculation and application. Unfortunately in later stages of the optimization these diagonal preconditioners are not as well suited for the inner iteration as they are for the optimization in the earlier stages. This is mostly due to an increased number of elements of the dense Hessian having the same order of magnitude near an optimum.

We construct a sparse approximate inverse preconditioner (SPAI) that is used to accelerate the inner iteration especially in the later stages of the optimization. The quality of our preconditioner depends on a predefined sparsity pattern. We apply the knowledge of the pattern of the Gauss-Newton approximation of the Hessian to efficiently construct the needed pattern for our preconditioner which can then be computed efficiently fully in parallel using GPUs. This preconditioner is then applied to a deep auto-encoder test case using different update strategies.

Contents

Abstract	i
Outline of the Thesis	iv
I. Introduction	1
II. Theory	5
1. Iterative Solution of Linear Equation Systems	6
1.1. Stationary Iterative Methods	6
1.1.1. Jacobi Method	7
1.1.2. Gauss-Seidel Method	7
1.2. Nonstationary Iterative Methods	7
1.2.1. Conjugate Gradient Method	8
1.2.2. GMRES Method	9
2. Preconditioning	11
2.1. Implicit preconditioners	11
2.2. Explicit preconditioners	11
3. Hessian-free Optimization	14
3.1. Truncated Newton Methods	14
3.2. Levenberg-Marquardt Method	15
3.3. Implemented algorithm	16
3.4. Hessian vs Gauss-Newton approximation	16
3.5. Inner iteration of the HF method	17
3.6. Preconditioning of inner iteration	19
3.7. Peculiarities of Martens implementation	20
4. Neural networks	21
4.1. Feed-forward Networks	21
4.2. Deep auto-encoder	23
4.3. Network Training	24
4.4. Error backpropagation	25
4.5. Fast multiplication by the Hessian	27

4.6. Hessian and Gauss-Newton Hessian for neural networks	29
III. Construction of the SPAI Preconditioner	31
5. Efficient pattern finding	33
5.1. Gauss-Newton approximation	34
5.2. Building the sparsity pattern	34
5.3. Summary of the pattern finding method	35
IV. Case study	37
6. Training of a deep auto-encoder	38
6.1. Overview	38
6.2. Pathological Curvature	39
6.3. Analysis	40
6.4. Results for different sparsity pattern	44
6.5. Results for reduced models	46
6.6. Results using CG for the inner iteration	48
V. Conclusions and Future Work	51
7. Conclusions	52
8. Future Work	53
8.1. Parallelization of the method	53
8.2. Termination of CG	53
8.3. Application of a better approximation of the diagonal of the Hessian . .	53
8.4. Using more information for the preconditioner	54
8.5. Efficient calculation of particular entries of the Hessian	54
Appendix	57
A. Pattern finding idea	57
B. Sparsity Patterns	58
C. Results for different Hessians	60
D. CG results	62
Bibliography	63

Outline of the Thesis

Part I: Introduction

This chapter presents an overview of the thesis and its purpose.

Part II: Theory

CHAPTER 1: ITERATIVE SOLUTION OF LINEAR EQUATION SYSTEMS

This chapter presents the basic theory for the iterative solution of linear equation systems. The first part deals with stationary methods like the Jacobi and Gauss-Seidel method. The second part deals with Krylov subspace methods which are nonstationary methods that form the basis for the inner iteration of truncated Newton methods. Examples are the Conjugate Gradient method and GMRES.

CHAPTER 2: PRECONDITIONING

How fast iterative methods like CG find a solution depends on spectral properties of the underlying matrix. It is often convenient to transform the equation system into a system that has the same solution but has more favorable spectral properties. This is achieved by finding a suitable preconditioner. We will focus on a sparse preconditioner that is an approximation of the inverse of A . The main task is to find an appropriate sparsity pattern of A^{-1} without explicit knowledge.

CHAPTER 3: HESSIAN-FREE OPTIMIZATION

This chapter summarizes the Hessian-Free or Truncated Newton method used for the optimization of nonlinear functions. It presents basic ideas and discusses peculiarities in Martens implementation used for training of deep neural networks. A particular focus lies on the stopping criteria of the inner iteration of the optimization method.

CHAPTER 4: NEURAL NETWORKS

This chapter presents the basics of neural networks that are needed to understand the theory behind the case study. It deals with training of neural networks, the backpropagation algorithm used to compute gradient information needed for training of the net and a variant known as the “Pearlmutter trick” used to efficiently compute the matrix vector product of the Hessian $\mathbf{H}\mathbf{v}$ with an arbitrary vector.

Part III: Construction of the SPAI Preconditioner

This part describes the straightforward approach of Benzi to compute a feasible a priori sparsity pattern for a sparse approximate inverse preconditioner.

CHAPTER 5: PATTERN FINDING

Because of the sparsification costs of the above method it is not economic to apply this method to find a suitable pattern for our preconditioner. This chapter presents an approach to come up with a suitable pattern by exploiting the known pattern of the Gauss-Newton Hessian that is usually used as a substitute of the Hessian in the context of the Hessian-Free optimization method.

Part IV: Case study

CHAPTER 6: TRAINING OF A DEEP AUTO-ENCODER

In this case study Martens Matlab implementation of the Hessian-Free optimization method is used to train a deep auto-encoder. This task is known to be hard for algorithms based only on gradient information like the backpropagation algorithm, especially without pretraining.

Then the changing structure of the Gauss-Newton Hessian in the different stages of the optimization is investigated and results of applying different preconditioners on representatives of the different stages are presented.

Furthermore the final results of the application of the SPAI preconditioner, using different update strategies in HF using GMRES or CG for the inner iteration with different stopping criteria based on the value of the quadratic approximation, are discussed .

Part V: Conclusions and Future Work

CHAPTER 7: CONCLUSIONS

This chapter summarizes the thesis.

CHAPTER 8: FUTURE WORK

This chapter presents some possible topics for future work like studies dealing with speedup of the computation of SPAI preconditioners on a graphics card or a possibly more efficient way to compute the needed entries of the Hessian for the computation of our preconditioner.

Part I.

Introduction

Introduction

Recently there are a lot of algorithms trying to incorporate second order information into the training of deep neural networks. The task of training these networks is known to be very hard using common training algorithms like backpropagation.

This is mostly due to two reasons. The first one is the well known vanishing gradients problem and the second one is pathological curvature. The latter is making it very tough for algorithms based only on gradient information.

It often appears that they get stuck in a local minimum because of pathological curvature.

Following this reasoning, Martens (2010) [25] successfully applied a variant of the Hessian-free optimization algorithm to the task of learning for deep auto-encoders and more recently also to recurrent neural networks [26].

The Hessian-free optimization method, better known as truncated-newton method, uses the idea of Newton's method and some globalisation ideas to find the minimum of a nonlinear function. Other than Newton's method it settles for solving the Newton equations only approximately using some iterative Krylov subspace method like the Conjugate Gradient method. Thus the explicit knowledge of the Hessian is not needed. Instead there is only the need to efficiently compute matrix-vector products of the Hessian with arbitrary vectors. This is possible through the well-known Pearlmuttertrick [30], which uses a variant of the backpropagation algorithm for feed forward neural networks.

There is also an approach called Krylov subspace decent [34] that explicitly computes a basis of the Krylov subspace for a fixed $k < N$ and numerically optimizes the parameter change within this subspace, using BFGS to minimize the original nonlinear objective function.

Martens uses a handmade diagonal preconditioner to speed up the inner CG iteration based on some approximate values of the diagonal entries of the Hessian.

A further improvement of the HF method of Martens is the use of a better diagonal preconditioner using the exact values of the diagonal of the Hessian as supposed in [9]. All these methods have in common that they use an easy to compute diagonal preconditioner for the inner iteration.

We will abandon the restriction of diagonal preconditioning and apply a sparse approximate inverse (SPAI) preconditioner [19],[21] to the Krylov subspace method. This approach has the advantages that the computation can be done completely in parallel and that the preconditioned residuals in the CG algorithm can be obtained by a simple matrix vector product instead of being the solution of a linear equation system.

As we will see, the choice of diagonal preconditioners makes perfect sense in the early stages of the optimization, in particular if a Levenberg-Marquardt heuristic, as in Martens method, is used to guarantee global convergence of the HF method.

The shifted equation system in outer iteration k is given by $(\mathbf{H}_k + \lambda_k \mathbf{I})p_k = -g_k$.

In the early stages of the optimization, λ will be greater than zero and the diagonal entries will be the dominating entries, therefore it should be sufficient for a preconditioner to capture these entries.

But in later stages, in particular near a minimum, λ will be almost zero and there will be a lot of elements in \mathbf{H} that have the same order of magnitude.

Benzi successfully applied a SPAI preconditioner to speed up Krylov subspace methods even if the considered matrices are dense [5]. This is also the case for our Hessian.

It is vital for the success of the preconditioner to capture the positions of the largest elements of A because they tend to also be positions of the largest elements of A^{-1} .

We first follow Benzi's approach to find an optimal pattern for our preconditioner for some Hessians obtained in the process of optimization. Then we compare the performance of our SPAI preconditioner for different drop tolerances, and therefore different sparsity patterns, and show their superiority to the performance of other common preconditioners like ILU, Jacobi or Gauss-Seidel, even though most of them are clearly unpractical for the HF method.

Then we discuss how we can use cheaply available information in the HF method to come up with a good guess for the needed sparsity pattern of our preconditioner using knowledge how the Gauss-Newton approximation of the Hessian is built and compare the performance of our preconditioner for a fixed sparsity.

With this simple and efficient method to find a good sparsity pattern for our preconditioner for a predefined sparsity percentage or number of allowed nonzero elements at hand, the preconditioner is incorporated in Martens HF implementation.

For simplicity it is first used together with the GMRES method to test different update strategies.

Because of the comparable high costs to compute the preconditioner, especially if it is not possible to distribute the computation on a graphics card, it seems economical to use the same SPAI preconditioner for consecutive iterations.

We use Martens HF implementation and his standard testcase, the training of the CURVES dataset of a deep auto-encoder, to compare the different update strategies with the unpreconditioned and the diagonally preconditioned variant.

After the successful tests using GMRES instead of CG, we have a closer look at different stopping criteria for CG in the context of truncated-newton methods.

To only monitor the residual $r = b - Ax$ in the CG method is in general a bad idea if someone wants to stop the iteration before the iterative method actually finds the desired solution.

This is because CG minimizes a quadratic approximation instead of solving the equa-

tion system directly. Thus the value of the quadratic function is monotonically decreasing but not the residual.

Therefore it is better to use a stopping criteria that monitors the value of the quadratic approximation. Martens uses a handmade criteria to stop the inner iteration in [25].

Part II.

Theory

1. Iterative Solution of Linear Equation Systems

In many applications linear equation systems are obtained after discretization of a problem. Thus it is often enough to compute only an approximate solution in the already introduced order of the discretization error. An approximate solution is usually computed way faster than the exact solution.

An iterative method finds an infinite sequence of approximate solutions to the exact solution of the linear system. In the iteration process the error is usually decreased by using A repeatedly and without modifying.

The idea is to produce such a sequence by applying an iterative function Φ .

$$x^{k+1} = \Phi(x^k) \Rightarrow x^k \xrightarrow{k \rightarrow \infty} \bar{x} = A^{-1}b. \quad (1.1)$$

First we introduce Jacobi and Gauss-Seidel methods as examples of stationary iterative methods.

Then we investigate the Conjugate Gradient method and GMRES as examples of non-stationary iterative methods. We follow [1].

1.1. Stationary Iterative Methods

These methods can be written in two normal forms.

- $x^{k+1} = x^k + Fr_k$ with preconditioner F and residual $r_k = b - Ax^k$, or
- $x^{k+1} = c + Bx^k$, where $B = I - FA$.

The first form formulates the iterative method as a residual based correction scheme. The second one states that x^{k+1} is obtained by a linear transformation of the previous iterate.

Starting from the initial value $x^0 = 0$, the solution at iteration k will lie in the Krylov-space $K_k(B, c)$.

$$x^{k+1} \in K_k(B, c) := \text{span} \{c, Bc, B^2c, \dots, B^{k-1}c\} \quad (1.2)$$

Stationary methods differ in the choice of the iteration matrix B resp. preconditioner F .

1.1.1. Jacobi Method

To obtain the Jacobi method, we first write the equation system row-wise.

$$\sum_{j=1}^n a_{ij}x_j = b_i$$

Then we solve the equation in row i for x_i while assuming that the other entries of x remain fixed.

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij}x_j \right)$$

This suggests an iterative method

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij}x_j^{(k)} \right)$$

In matrix terms, the Jacobi method can be expressed as

$$x^{(k+1)} = D^{-1}(L + U)x^{(k)} + D^{-1}b$$

where the matrices D , $-L$, $-U$ represent the diagonal, the strictly lower-triangular, and the strictly upper-triangular parts of A .

1.1.2. Gauss-Seidel Method

If we proceed as with the Jacobi method, but now assume that the equations are examined one at a time in sequence, and that previously computed results are used as soon as they are available, we obtain the Gauss-Seidel method:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j < i} a_{ij}x_j^{(k+1)} - \sum_{j > i} a_{ij}x_j^{(k)} \right)$$

$$x^{(k+1)} = (D - L)^{-1}(Ux^{(k)} + b) = x^{(k)} + (D - L)^{-1}r_k$$

where the matrices D , $-L$, $-U$ represent the diagonal, the strictly lower-triangular, and the strictly upper-triangular parts of A .

The convergence of these methods depends on the spectral radius of the iteration matrix B , i.e. $\rho(B) < 1$. This is for example true for strictly diagonal dominant matrices.

1.2. Nonstationary Iterative Methods

Nonstationary methods differ from stationary methods in that the computations involve information that changes at each iteration. Typically, constants are computed by taking inner products of residuals or other vectors arising from the iterative method.

1.2.1. Conjugate Gradient Method

The Conjugate Gradient method is an effective method for symmetric positive definite systems, thus it is the perfect candidate to approximately solve the Newton equation in a truncated Newton method.

Instead of solving $Ax = b$, CG minimizes the quadratic $\Phi(x)$

$$\Phi(x) = \frac{1}{2}x^T Ax - x^T b.$$

The approximate solutions are updated by the formula

$$x^{(k)} = x^{(k-1)} + \alpha_k p^{(k)}, \quad (1.3)$$

where α_k is the stepsize and $p^{(k)}$ the search direction.

The stepsize α_k is found by a one-dimensional minimization of $\Phi(x^{(k)} + \alpha_k p^{(k)})$ as

$$\alpha_k = \frac{r^{(k-1)T} r^{(k-1)}}{p^{(k)T} A p^{(k)}}, \quad (1.4)$$

where the residual $r^{(k)} = b - Ax^{(k)}$ is updated as

$$r^{(k)} = r^{(k-1)} - \alpha_k q^{(k)}, \quad (1.5)$$

where $q^{(k+1)} = A p^{(k+1)}$.

Then the search direction is updated using the residuals

$$p^{(k)} = r^{(k)} + \beta_{k-1} p^{(k-1)}, \quad (1.6)$$

where the choice $\beta_k = \frac{r^{(k)T} r^{(k)}}{r^{(k-1)T} r^{(k-1)}}$ ensures that $p^{(k)}$ and $A p^{(k-1)}$ are orthogonal.

In fact, the choice of β_k ensures that $p^{(k)}$ and $r^{(k)}$ are orthogonal to all previous $A p^{(j)}$ and $r^{(j)}$ respectively.

Since for symmetric A an orthogonal basis for the Krylov subspace $\text{span}\{r^{(0)}, \dots, A^{i-1} r^{(0)}\}$ can be constructed with only three-term recurrences, such a recurrence also suffices for generating the residuals. In the Conjugate Gradient method two coupled two-term recurrences are used, one that updates residuals using a search direction vector. and one updating the search direction with a newly computed residual. This makes the Conjugate Gradient Method computationally attractive.

Listing 1.1 shows an implementation of the preconditioned algorithm.

Note that to get fast convergence and reduce the number of iterations, it is often needed to apply a suitable preconditioner M , such that $M^{-1}Ax = M^{-1}b$ has more favorable spectral properties, i.e. clustered eigenvalues.

The preconditioned CG method using preconditioner $M = LL^T$ minimizes the following quadratic form.

$$\varphi_2(y) = \frac{1}{2}y^T L^T A L y - y^T L^T b,$$

```

 $r^{(0)} = b - Ax^{(0)}$  for initial guess  $x^{(0)}$ 
for  $k = 1, 2, \dots$ 
    solve  $Mz^{(k-1)} = r^{(k-1)}$ 
     $p^{(k-1)} = r^{(k-1)T}z^{(k-1)}$ ;
    if  $k == 1$ 
         $p^{(1)} = z^{(0)}$ ;
    else
         $\beta_{k-1} = p^{(k-1)} / p^{(k-2)}$ ;
         $p^{(k)} = z^{(k-1)} + \beta_{k-1}p^{(k-1)}$ ;
    end
     $q^{(k)} = Ap^{(k)}$ ;
     $\alpha_k = \frac{p^{(k-1)}}{p^{(k)T}q^{(k)}}$ ;
     $x^{(k)} = x^{(k-1)} + \alpha_k p^{(k)}$ ;
     $r^{(k)} = r^{(k-1)} - \alpha_k q^{(k)}$ 
    if  $\|r^{(k)}\| < \varepsilon$ 
        return ;
    end
end

```

Listing 1.1: The Conjugate Gradient method

where $y = L^{-1}x$.

After n steps of the CG method $K_n(A, b) = \mathbb{R}^n$ and therefore $x_n = A^{-1}b$ is the solution in exact arithmetic. This is not true in floating point arithmetic.

For CG, the error can be bounded in terms of the spectral condition number of the matrix $M^{-1}A$. If \bar{x} is the exact solution of the linear system $Ax = b$, with symmetric positive definite matrix A , then for CG, it can be shown that

$$\|x^{(k)} - \bar{x}\|_A \leq 2\gamma^k \|x^{(0)} - \bar{x}\|_A$$

where $\gamma = \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$ and $\kappa = \kappa(M^{-1}A) = \frac{\lambda_{max}}{\lambda_{min}}$.

Note that CG tends to eliminate components of the error in the direction of eigenvectors associated with extremal eigenvalues first. After these have been eliminated, the method proceeds as if these eigenvalues did not exist in the given system, i.e. the convergence rate depends on a reduced system with a smaller condition number. See [33] for an analysis of this.

It is possible to compute a preconditioner for the CG method that makes use of this. See Section 8.4 for more information.

1.2.2. GMRES Method

In the Conjugate Gradients method, the residuals form an orthogonal basis for the space $\text{span}\{r^{(0)}, Ar^{(0)}, A^2r^{(0)}, \dots\}$.

In GMRES, this basis is formed explicitly.

```

 $w^{(i)} = Av^{(i)}$ 
for  $k = 1, \dots, i$ 
     $w^{(i)} = w^{(i)} - (w^{(i)}, v^{(k)})v^{(k)}$ 
end
 $v^{(i+1)} = \frac{w^{(i)}}{\|w^{(i)}\|}$ 

```

Listing 1.2: Orthogonal basis of the Krylov space

This is a modified Gram-Schmidt orthogonalization. Applied to the Krylov sequence $\{A^k r^{(0)}\}$ this orthogonalization is called the Arnoldi method. The inner product coefficients $(w^{(i)}, v^{(k)})$ and $\|w^{(i)}\|$ are stored in an upper Hessenberg matrix.

The GMRES iterates are constructed as

$$x^{(i)} = x^{(0)} + y_1 v^{(1)} + \dots + y_i v^{(i)},$$

where the coefficients y_k have been chosen to minimize the residual norm $\|b - Ax^{(i)}\|$. Note that, while the CG method is both, cheap and optimal, GMRES is only an optimal method. With increasing iterations the computation and storage of the orthogonal basis of the Krylovspace gets very expensive. Therefore one often settles with a restarted variant of GMRES.

2. Preconditioning

The convergence rate of iterative methods depends on spectral properties of the coefficient matrix. Hence one may attempt to transform the linear system into one that is equivalent in the sense that it has the same solution, but has more favorable spectral properties. A preconditioner is a matrix that effects such a transformation.

For instance, if a matrix M approximates the coefficient matrix A in some way, the transformed system

$$M^{-1}Ax = M^{-1}b$$

has the same solution as the original system $Ax = b$, but the spectral properties of its coefficient matrix $M^{-1}A$ may be more favorable.

In devising a preconditioner, we are faced with a choice between finding a matrix M that approximates A , and for which solving a system is easier than solving one with A , or finding a matrix M that approximates A^{-1} , so that only multiplication by M is needed.

The majority of preconditioners falls in the first category, but the preconditioner we will use will be an approximation of A^{-1} .

2.1. Implicit preconditioners

These preconditioners try to approximate A . Theoretically the perfect preconditioner is $M = A^{-1}$, or thinking about the LU decomposition of A , we want to choose $M = (LU)^{-1}$. Of course it is too expensive to actually compute the inverse and then use it as preconditioner.

Nevertheless we can use an approximation of the LU decomposition as implicit preconditioner of A .

The idea is to let L, U take nonzero values only at nonzero positions of A , i.e. keep the sparsity pattern of A and $L + U$ the same. This is the so-called $ILU(0)$, with no fill-ins allowed. The algorithm is shown in listing 2.1.

2.2. Explicit preconditioners

These types of preconditioners try to approximate A^{-1} . Implicit preconditioners are robust and lead to fast convergence of the preconditioned iteration but are difficult to implement in parallel. In particular the triangular solves involved in ILU preconditioning represent a serious bottleneck, thus limiting the performance on parallel computers.

```

for  $j = 1, \dots, n$ 
  for  $i = j + 1, \dots, n$ 
    if  $A_{ij} \neq 0$ 
       $A_{ij} = A_{ij}/A_{jj};$ 
      for  $k = j + 1, \dots, n$ 
        if  $A_{ik} \neq 0$ 
           $A_{ik} = A_{ik} - A_{ij}A_{jk};$ 
        end;
      end;
    end;
  end;
end;

```

Listing 2.1: ILU(0) method

The explicit approach has some advantages over the implicit approach. First the preconditioner can be computed fully in parallel and second if used as preconditioner of the PCG method there is no equation system $My = r$ to be solved to get the preconditioned residual. Instead only one matrix-vector multiplication $y = M \cdot r$ is needed. These computations are easier to parallelize than the triangular solves needed for implicit preconditioning. This is why inverse preconditioners are becoming increasingly popular.

We will only consider one special type of preconditioner, the sparse approximate inverse preconditioner (SPAI). For more information, see [19], [21].

Sparse Approximate Inverse (SPAI) Preconditioner

We can think of a SPAI preconditioner as the best matrix that has a prescribed sparsity pattern S and minimizes the following functional (in the Frobenius norm).

$$\min_M \|AM - I\|_F^2 \quad (2.1)$$

The Frobenius norm naturally decouples (2.1) into n Least Squares problems

$$\min_{m_j} \|Am_j - e_j\|_2^2, \quad j = 1, \dots, n \quad (2.2)$$

where $M = [m_1 \dots m_n]$, $I = [e_1 \dots e_n]$.

Thus the computation is embarrassingly parallel.

Due to the sparsity of A , equation (2.2) represents an usually small-sized Least-squares problem to solve, although n can be very large. Even if A is sparse, A^{-1} will usually not be sparse. The main issue is the selection of the nonzero pattern of M . There are different possibilities we can choose from to create our a priori sparsity pattern.

Different choices for the sparsity pattern of $M \approx A^{-1}$

We could use the pattern of

- A^k or $(A^T)^k$
- $(A^T A)^k A^T$ for some $k = 1, 2$
- A_ε with sparsified A

where A_ε is obtained by deleting all entries with $|A_{i,j}| \leq \varepsilon$.

Following Benzi, [5] an easy way to compute a sparsity pattern of M is the use of a simple criterion based on a threshold parameter $\varepsilon \in (0, 1)$ and to include position (i, j) in the nonzero pattern of M if

$$|a_{ij}| > \max_{1 \leq k, l \leq n} |a_{kl}|. \quad (2.3)$$

For our deep auto encoder test case, we will compute A explicitly and then sparsify A using different drop tolerances ε_i at different iterates to get some insight into the changing structure of the Gauss-Newton approximation of the Hessian used in the HF algorithm.

This procedure is very expensive and clearly unpractical to use for the pattern finding part in the HF algorithm.

Fortunately we can use cheaply available information about the different magnitudes of the diagonal elements of the Gauss-Newton approximation to find a good sparsity pattern for our preconditioner without explicit knowledge of A .

Moreover we are able to find the best pattern for a predefined sparsity percentage of A . We will focus on this in Chapter III.

In the context of the Hessian-Free optimization algorithm, the Gauss-Newton Hessian is used instead of the Hessian. This approximation is always symmetric positive definite, therefore it is reasonable to compute our SPAI preconditioner in a factorized form $M = LL^T$. Hence FSPAI, a factorized version of the preconditioner, is used in the tests. See Huckle [20] for more information.

3. Hessian-free Optimization

The Hessian-Free optimization method, also known as truncated Newton method, tries to minimize a nonlinear function f . But instead of computing the exact Newton direction $-\mathbf{H}^{-1}g$ where \mathbf{H} is the Hessian and g the gradient, a truncated Newton method solves the system $\mathbf{H}d = -g$ only approximately using e.g. the conjugate gradient method. CG is solving the system using only matrix-vector multiplications Hv . These multiplications can be done efficiently using a technique known as the “Pearlmutter-trick”.

The Hessian is never computed explicitly and is only available implicitly through matrix-vector products. Thus the name “Hessian-Free”.

Martens adapted the Hessian-Free method for the training of deep auto-encoders.

This chapter presents the idea of the Hessian-Free optimization method in some detail. Then the subtleties in Martens implementation are discussed. Furthermore we compare the differences of the Hessian and the Gauss-Newton Hessian and why it is preferable to use the latter one. In the end, we look at different stopping criteria for the inner iteration.

3.1. Truncated Newton Methods

The Hessian-free optimization method is a second order quasi-newton method that optimizes a given nonlinear function $f(\mathbf{w})$. It doesn’t use a low rank approximation of the Hessian. Only the matrix-vector product of the Hessian with a vector d is needed for the optimization, thus the Hessian is never computed explicitly. Fortunately there exists an efficient algorithm for neural networks that computes these matrix-vector products exactly and efficiently in $O(\mathbf{w})$, where \mathbf{w} are the weights of the neural network that has to be trained.

Truncated-Newton methods are a family of methods suitable for solving large nonlinear optimization problems. At each iteration, the current estimate of the solution is updated (i.e., a step is computed) by approximately solving the Newton equations using an iterative algorithm. This results in a doubly iterative method: an outer iteration for the nonlinear optimization problem, and an inner iteration for the Newton equations. The inner iteration is typically stopped or “truncated” before the solution to the Newton equations is obtained. More generally, an “inexact” Newton method computes a step by approximately solving the Newton equations.

We focus on the unconstrained problem

$$\min_{\mathbf{w}} E(\mathbf{w}) \tag{3.1}$$

where the nonlinear function $E(\mathbf{w})$ is for example the sum-of-squares reconstruction error of our deep auto-encoder.

The first-order optimality condition for this problem is

$$\nabla E(\mathbf{w}) = 0 \quad (3.2)$$

which is a system of nonlinear equations.

Given some guess w_k of a solution \tilde{w} , Newton's method computes a step p_k as the solution to the linear system.

$$\nabla^2 E(w_k) p_k = -\nabla E(w_k) \quad (3.3)$$

and then sets $w_{k+1} = w_k + p_k$. In this simple form, Newton's method is not guaranteed to converge.

In a truncated-Newton method, an iterative method is applied to (3.3), and an approximate solution accepted.

The rate of convergence of the outer iteration is related to the accuracy with which (3.3) is solved.

A basic question in a truncated-Newton method is the choice of an inner iterative algorithm for solving (3.3). Some variant of the linear conjugate-gradient method is almost always used. The conjugate-gradient method is an optimal iterative method for solving a positive-definite linear system $Ap = b$, in the sense that the k th iterate p_k minimizes the associated quadratic function $\mathcal{Q}(p) = \frac{1}{2}p^T Ap - p^T b$ over the Krylov subspace spanned by $\{b, Ab, \dots, A^{k-1}b\}$.

The Hessian matrix $\nabla^2 E(w_k)$ need not be positive definite, so the assumptions underlying the conjugate-gradient method may not be satisfied. However, the Hessian matrix is always symmetric. At a local minimizer of (3.2), the Hessian is guaranteed to be positive semi-definite. Thus, as the solution is approached, and the Newton model for (3.2) is more accurate and appropriate, we can anticipate that the requirements for the conjugate-gradient method will be satisfied.

A truncated-Newton method will only be competitive if further enhancements are used. For example, a preconditioner for the linear system will be needed, and the stopping rule for the inner algorithm will have to be chosen in a manner that is effective both close to and far from the solution. With these enhancements, truncated-Newton methods are a powerful tool for large-scale optimization.

3.2. Levenberg-Marquardt Method

The Levenberg-Marquardt method is a compromise between the following two methods:

- Newton's method, which converges rapidly near a local or global minimum, but may also diverge

- Gradient descend, which is assured to converge through a proper selection of the step-size parameter, but converges slowly.

Consider the modified Newton equations for outer iteration k .

$$(\nabla^2 E(w_k) + \lambda \mathbf{I})p_k = -\nabla E(w_k) \quad (3.4)$$

where \mathbf{I} is the identity matrix of the same dimensions as $\mathbf{H} = \nabla^2 E(w_k)$ and λ is a regularizing parameter that forces the sum matrix $(\mathbf{H} + \lambda \mathbf{I})$ to be positive definite and savely well conditioned throughout the computation.

When λ is large, $\lambda \mathbf{I}$ dominates \mathbf{H} , the method reduces to the gradient descend method. If λ is small the estimated increment is almost the same as that of the Gauss-Newton method. The value of λ varies at each iteration, depending on whether the objective function decreases.

Martens uses a simple heuristic for adjusting λ .

If $\rho < \frac{1}{4}$: $\lambda \leftarrow \frac{3}{2}\lambda$, if $\rho > \frac{3}{4}$: $\lambda \leftarrow \frac{2}{3}\lambda$, where ρ is the “reduction ratio”.

The reduction ratio is a scalar quantity which attempts to measure the accuracy of the quadratic model Q and is given by:

$$\rho = \frac{E(\mathbf{w} + p) - E(\mathbf{w})}{Q(p) - Q(0)}. \quad (3.5)$$

3.3. Implemented algorithm

```

for n = 1,2,... do
   $g_n \leftarrow \nabla f(w_n)$  compute/adjust  $\lambda$  by some method
  define the function  $B_n(d) = \mathbf{H}(w_n)d + \lambda d$ 
   $p_n \leftarrow \text{CG-Minimize}(B_n, -g_n)$ 
   $w_{n+1} \leftarrow w_n + p_n$ 
end for

```

Listing 3.1: The Hessian-Free Optimization Algorithm

3.4. Hessian vs Gauss-Newton approximation

The training error of a neural network is in general a non-convex function, therefore the Hessian \mathbf{H} might not be positive definite, so Newtons method may diverge. Thus most algorithms use approximations of the Hessian that remain positive definite. One choice is the use of the Gauss-Newton approximation of the Hessian. Sometimes it is called linearized, outer product or squared Jacobian approximation.

The idea is to approximate the objective function E around w_k by a quadratic function. Specifically the Newton algorithm uses the second-order Taylor expansion

$$W(w_k + p) \approx W(w_k) + p^T \nabla W(w_k) + \frac{1}{2} p^T \mathbf{H}(w_k) p := \mathcal{Q}^k(p) \quad (3.6)$$

The second order Taylor expansion is called the Newton approximation of the function E , as it is related to the Newton algorithm.

Similarly when \mathbf{H} is replaced by the Gauss-Newton matrix \mathbf{H}_{GN} , the quadratic approximation is called Gauss-Newton approximation.

The Gauss-Newton Hessian is introduced for neural networks in Section 4.6.

3.5. Inner iteration of the HF method

The inner iteration of a truncated Newton method almost always consists of a variant of the Conjugate Gradient algorithm. Additionally to the common residual based stopping criterion, there is also a stopping criterion used that depends on the optimization progress of the quadratic model.

Algorithm: Truncated Newton (Inner Loop of Outer Step k)

0. Initialization.

Set $p^0 = 0$, $q^0 = q_k(p^0) = 0$, $r_0 = -g$, and $d_0 = Mr_0$.

For $i = 0, 1, 2, \dots$ proceed as follows:

1. Negative Curvature Test

If $d_i^T \mathbf{H} d_i < 0$,
exit inner loop

2. Truncation Test.

$$\alpha_i = \frac{r_i^T (Mr_i)}{d_i^T \mathbf{H} d_i}$$

$$p^{i+1} = p^i + \alpha_i d_i$$

$$r_{i+1} = r_i - \alpha_i \mathbf{H} d_i$$

$$q^{i+1} = \frac{1}{2} (r_{i+1} + g)^T p^{i+1} \quad \text{Value of } \mathcal{Q}(p^{i+1})$$

If $\text{STOP}_{Residual}$ is satisfied for r_{i+1} , or $\text{STOP}_{Quadratic}$ is satisfied for $\mathcal{Q}(p^{i+1})$, exit inner loop with search direction $p = p^{i+1}$.

3. Continuation of PCG

$$\beta_i = \frac{r_{i+1}^T (Mr_{i+1})}{r_i^T (Mr_i)}$$

$$d_{i+1} = (Mr_{i+1}) + \beta_i d_i$$

Stopping Criteria for inner iteration

If the conjugate-gradient method is used for the inner iteration, then the i th inner iteration finds a minimizer of the quadratic model

$$f(w_k + p) \approx f(w_k) + p^T \nabla f(w_k) + \frac{1}{2} p^T \mathbf{H}(w_k) p := \mathcal{Q}^k(p) \quad (3.7)$$

over the Krylov subspace spanned by $\{\nabla f(w_k), \dots, [\mathbf{H}(w_k)]^{i-1} \nabla f(w_k)\}$.

The model (3.7) has a global minimum when the residual of the Newton equations is zero. If p is not the minimizer of the quadratic model, however, the magnitudes of the residual and the quadratic model can be dramatically different, and the residual can be a deceptive measure of the quality of the search direction.

Figure 3.1 shows the residual and the value of the quadratic $\mathcal{Q}(p) = \frac{1}{2} p^T A p - p^T b$ for a CG run in the latter stages of optimization.

The original motivation for running CG in the HF method was to minimize the quadratic, therefore the stopping criterion should be based on the change in the quadratic.

Let p_i be the search direction computed at the i -th inner iteration, and let $\mathcal{Q}_i = \mathcal{Q}(p_i)$. The stopping rule suggested in [28] is to accept a search direction if $i(\mathcal{Q}_i - \mathcal{Q}_{i-1})/\mathcal{Q}_i \leq \eta_k$. This criterion compares the reduction in the quadratic model at the current iteration ($\mathcal{Q}_i - \mathcal{Q}_{i-1}$) with the average reduction per iteration \mathcal{Q}_i/i . If the current reduction is small relative to the average reduction, then the inner iteration is terminated.

Newton's method is based on the Taylor series approximation (3.7). If this approximation is inaccurate then it may not be sensible to solve the Newton equations accurately. "Over-solving" the Newton equations will not produce a better search direction. If this is the case, CG should be terminated after a few steps.

Martens elects to stop CG once the relative per-iteration progress made in minimizing $\mathcal{Q}(p)$ fell below some tolerance. He terminates CG at iteration i if:

$$i > k \quad \text{and} \quad \mathcal{Q}(p_i) < 0 \quad \text{and} \quad \frac{\mathcal{Q}(p_i) - \mathcal{Q}(p_{i-k})}{\mathcal{Q}(p_i)} < k\varepsilon \quad (3.8)$$

where k determines how many iterations into the past we look in order to compute an estimate of the current per-iteration reduction rate. ($k = \max(10, 0.1i)$ and $\varepsilon = 0.0005$). Thus averaging over a progressively larger interval as i grows.

Martens states the following: "This approach causes CG to terminate in very few iterations when λ is large which is the typical scenario during the early stages of optimization.

In later stages $\mathcal{Q}(p)$ begins to exhibit pathological curvature, which reflects the actual properties of f , thus making it harder to optimize. In these situations the termination condition will permit

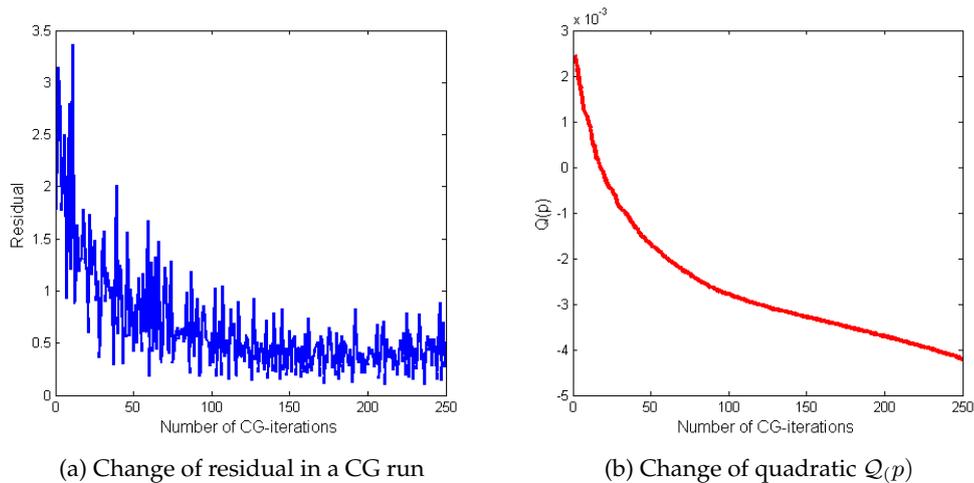


Figure 3.1.: Typical behaviour of residual $\|Ax - b\|$ and value of the quadratic $Q(p)$

CG to run much longer, resulting in a much more expensive HF iteration. But this is the price that seemingly must be paid."

3.6. Preconditioning of inner iteration

The convergence of the conjugate-gradient method is strongly influenced by the condition number of the Hessian (i.e., its extreme eigenvalues), and by the number of distinct eigenvalues of the Hessian. Reducing either of these accelerates the convergence of the method. Ideally, a preconditioner will be chosen based on the problem being solved. This can require considerable analysis and programming to accomplish, however, and is not suitable for routine cases. If the Hessian matrix is available, a good 'generic' choice of a preconditioner is an incomplete Cholesky factorization. The preconditioner is formed by factoring the Hessian, and ignoring some or all of the fill-in that occurs during Gaussian elimination.

Martens chooses a handmade diagonal preconditioner M :

$$M = \left[\text{diag} \left(\sum_{i=1}^D \nabla f_i(\theta) \odot \nabla f_i(\theta) \right) + \lambda I \right]^\alpha \quad (3.9)$$

Where f_i is the value of the objective associated with training-case i , \odot denotes the element-wise product and the exponent α is chosen to be less than 1 in order to suppress 'extreme' values.

The inner sum has the interpretation of being the diagonal of the empirical Fisher information matrix, which is similar in some way to the Gauss-Newton Hessian.

It is impractical to use the diagonal of \mathbf{H}_{GN} itself, since the costs would be similar to K

backpropagation operations, where K is the size of the output layer, which is large for auto-encoders.

3.7. Peculiarities of Martens implementation

Computing the matrix-vector products

Pearlmutter showed that there exists an efficient procedure for computing the product Hd exactly for neural networks(see [30]). Because it is similar to the backpropagation algorithm, it involves forward and backward passes through the network and has a similar computational cost.

Schraudolph generalized Pearlmutter's method in order to compute the Gauss-Newton approximation of the Hessian(see [31]).

Furthermore he uses the Gauss Newton Hessian \mathbf{H}_{GN} instead of the Hessian \mathbf{H} because it is guaranteed to be positive semi-definite thus avoiding the problem of negative curvature in the CG algorithm.

Sharing information across iterations

Martens uses the search direction p_{n-1} found by CG in the previous HF iteration as the starting point for CG in the current one. The reasoning is that the values of \mathbf{H} and $\nabla f(w)$ for a given HF iteration should be similar in some sense to the previous ones.

Random initialization

Martens initializes the weights of the network using a sparse initialization scheme. The number of non-zero incoming connection weights to each unit are hard limited and the biases are set to 0 or 0.5 for tanh units. *"Doing this allows the units to be both highly differentiated as well es unsaturated, avoiding the problem in dense initializations where the connection weights must all be scaled very small in order to prevent saturation, leading to poor differentiation between units."*

4. Neural networks

One common approach for modelling, e.g. using finite elements, is to write the solution y of a PDE as a linear combination of fixed basis functions $y(x, w) = \sum_i w_i \varphi_i(x)$. Although this kind of models have useful analytical and computational properties, their practical applicability is limited by the curse of dimensionality.

Using neural networks we take an alternative approach. We fix the number of basis functions in advance but allow them to be adaptive, in other words we use parametric forms for the basis functions in which the parameter values are adapted during training.

We will be following Bishop [8] and only consider feed-forward neural networks.

We first introduce the basics of neural networks.

Then we focus on the construction of deep auto-encoders. The training of such an auto-encoder is used as a test-case of our preconditioner.

The common algorithm used to train a feed-forward neural network is the backpropagation algorithm introduced in Section 4.4.

It makes use of gradient information of the error function that we try to minimize.

Truncated Newton methods like HF also need a fast procedure to efficiently calculate the matrix vector product of the Hessian with arbitrary vectors to obtain information of the curvature of the error-surface. This information is needed for the inner CG iteration and are introduced in Section 4.5.

Section 4.6 shows how the Gauss-Newton Hessian is obtained for a neural network.

4.1. Feed-forward Networks

Neural networks use basic functions that are of the form

$$y_i(x, w) = f \left(\sum_{j=1}^M w_j \varphi_j(x) \right). \quad (4.1)$$

So each of the basis functions is itself a nonlinear function of a linear combination of the inputs, where the coefficients in the linear combination are adaptive parameters.

The basic neural network model can be described as a series of functional transfor-

mations. First M linear combinations of the input variables x_1, \dots, x_D in the form

$$a_j = \sum_{i=1}^D w_{ji}^{(1)} x_i + w_{j0}^{(1)} \quad (4.2)$$

are constructed, where $j = 1, \dots, M$, and the superscript (1) indicates that the corresponding parameters are in the first layer of the network.

The parameters $w_{ji}^{(1)}$ are called weights and the parameter $w_{j0}^{(1)}$ are biases. The quantities a_j are called activations.

Each of them is then transformed using a differentiable, nonlinear activation function $h(\cdot)$ to give

$$z_j = h(a_j). \quad (4.3)$$

These quantities correspond to the outputs of the basis functions, and are called hidden units.

The nonlinear functions are generally chosen to be sigmoidal functions such as logistic sigmoid or tanh function.

Figure 4.1a depicts a single neuron with its activation a_j and output z_j .

These values are again linearly combined to give output unit activations

$$a_k = \sum_{j=1}^M w_{kj}^{(2)} z_j + w_{k0}^{(2)} \quad (4.4)$$

where $k = 1, \dots, K$ and K is the total number of outputs.

Finally, the output unit activations are transformed using an appropriate activation function to give a set of network outputs y_k .

For standard regression problems, the activation function is the identity so that $y_k = a_k$.

Thus the neural network model is simply a nonlinear function from a set of input variables $\{x_i\}$ to a set of output variables $\{y_k\}$ controlled by a vector of weights w of adjustable functions.

$$y_k(x, w) = \sigma \left(\sum_{j=1}^M w_{kj}^{(2)} h \left(\sum_{i=1}^D w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \right) \quad (4.5)$$

This formula can be easily adapted to represent more general architectures using more than one hidden layer.

Neural networks are said to be universal approximators. For example, a two-layer network with linear outputs can uniformly approximate any continuous function on a compact input domain to arbitrary accuracy provided the network has a sufficiently large number of hidden units.

We focus on deep auto-encoders.

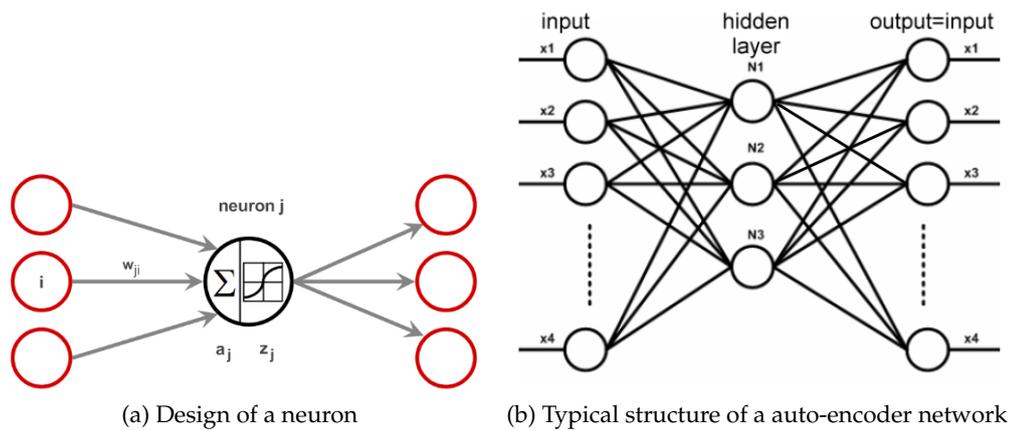


Figure 4.1.: Design of a single neuron and design of an auto-encoder

4.2. Deep auto-encoder

A auto-encoder is a special feed-forward neural network of the following design.

- The input and output layers have the same size m .
- The size of the hidden layer M is smaller than m .
- The network is fully connected.

The idea is to find a compact representation of the inputs in an unsupervised manner by letting the network recreate its own inputs and creating a bottleneck by using fewer hidden units than inputs.

A given pattern x is simultaneously applied to the input layer and to the output layer as the desired response.

The actual response of the output layer \hat{x} is intended to be an estimate of x .

By virtue of the special design of the network, the network is constrained to perform identity mapping through its hidden layer.

The activations of the hidden units are then used as a compact code for x . Figure 4.1b shows an auto-encoder with a single hidden layer.

A deep auto-encoder consists of multiple hidden layers and can be thought of as a nonlinear generalization of principal component analysis (PCA) that uses an adaptive, multilayer “encoder” network to transform the high-dimensional data into a low-dimensional code and a similar “decoder” network to recover the data from code.

Hinton and Salakhutdinov state the problem of training a deep auto-encoder as follows: *“It is difficult to optimize the weights in nonlinear autoencoders that have multiple hidden layers. With large initial weights, autoencoders typically find poor local minima; with small initial weights, the gradients in the early layers are tiny, making it infeasible to train autoencoders with*

many hidden layers. If the initial weights are close to a good solution, gradient descent works well, but finding such initial weights requires a very different type of algorithm that learns one layer of features at a time.” (cf. [18])

Thus expensive pretraining is used to come up with a good initialization for the weights of the deep network.

Martens [25] HF algorithm gives better results than the pretrained auto-encoders for different test cases even though he is using a random sparse initialization scheme for the weights. Note that the sum-of-squares reconstruction error function of a deep auto-encoder is in general non-convex and thus not easy to optimize.

4.3. Network Training

One approach to the problem of determining the network parameters \mathbf{w} is to minimize a sum-of-squares error function. Given a training set of input vectors $\{x_n\}$, where $n = 1, \dots, N$, together with a set of corresponding target values $\{t_n\}$, we minimize the error function

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N \|y(x_n, \mathbf{w}) - t_n\|^2. \quad (4.6)$$

This is equivalent to the maximization of the likelihood function obtained for a set of N independent, identically distributed observations $\mathbf{X} = \{x_1, \dots, x_N\}$, together with the target values $\mathbf{t} = \{t_1, \dots, t_N\}$.

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^N p(t_n|x_n, \mathbf{w}, \beta) = \mathcal{N}(t_n|y(x_n, \mathbf{w}), \beta^{-1}), \quad (4.7)$$

where we assumed that the target variable has a Gaussian distribution with an \mathbf{x} -dependent mean and precision β .

Taking the negative logarithm, we obtain the error function

$$\frac{\beta}{2} \sum_{n=1}^N \{y(x_n, \mathbf{w}) - t_n\}^2 - \frac{N}{2} \ln \beta + \frac{N}{2} \ln(2\pi). \quad (4.8)$$

Maximizing the likelihood function is equivalent to minimizing the sum-of-squares error function given by (4.6), where we have disregarded additive and multiplicative constants.

Because the error $E(\mathbf{w})$ is a smooth function of \mathbf{w} , its smallest value will occur at a point in weight space such that the gradient of the error function vanishes.

$$\nabla E(\mathbf{w}) = 0 \quad (4.9)$$

Otherwise we could make a small step in the direction of $-\nabla E(\mathbf{w})$ to further reduce the error.

Because there is no analytical solution to (4.9), iterative methods are used. Most of these techniques will update a given initial value \mathbf{w}^0 for the weight vector and then moving through weight space in a succession of steps of the form

$$\mathbf{w}^{k+1} = \mathbf{w}^k + \Delta \mathbf{w}^k \quad (4.10)$$

where k labels the iteration step.

We know that making a small step into the direction of the negative gradient $-\nabla E(\mathbf{w})$ will reduce the error. Therefore the simplest gradient based approach to reduce the error is the update

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \eta \nabla E(\mathbf{w}) \quad (4.11)$$

where the parameter $\eta > 0$ is known as the learning rate.

The error function is defined with respect to a training set, and so each step requires the entire training set to be processed in order to evaluate ∇E . Techniques that use the whole data set at once are called batch methods. For batch optimization, there are more efficient methods such as conjugate gradients, quasi-Newton or truncated Newton methods, which are much more robust and faster than simple gradient descent. The Hessian-Free method considered in the thesis is a truncated Newton method.

4.4. Error backpropagation

We will introduce the error backpropagation algorithm. This is a local message passing scheme in which information is sent alternately forwards and backwards through the network to efficiently compute the gradient $\nabla E(\mathbf{w})$.

Many error functions comprise a sum of terms, one for each data point in the training set, such that

$$E(w) = \sum_{n=1}^N E_n(\mathbf{w}). \quad (4.12)$$

The backpropagation algorithm efficiently evaluates $\nabla E_n(\mathbf{w})$ for one such term in the error function.

In a general feed-forward network, each unit computes a weighted sum of its inputs in the form

$$a_j = \sum_i w_{ji} z_i \quad (4.13)$$

where z_i is the activation of a unit, that sends a connection to unit j , and w_{ji} is the weight associated with that connection. Biases can be included by introducing an extra unit, or input, with activation fixed to +1.

The sum in (4.13) is transformed by a nonlinear activation function $h(\cdot)$ to give the activation z_j of unit j in the form

$$z_j = h(a_j). \quad (4.14)$$

Note that one or more of the variables z_i in the sum (4.13) could be an input, and the unit j could be an output.

For each pattern in the training set, we suppose that we have supplied the corresponding input vector to the network by successive application of (4.13) and (4.14). This process is called forward propagation.

Consider the evaluation of the derivative of E_n with respect to a weight w_{ji} . The outputs of the various units will depend on the particular pattern n . To keep the notation uncluttered, we omit the subscript n from the network variables.

First note that E_n depends on w_{ji} only via the summed input a_j to unit j .

We apply the chain rule for partial derivatives to give

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}. \quad (4.15)$$

Now set

$$\delta_j := \frac{\partial E_n}{\partial a_j} \quad (4.16)$$

where the δ 's are referred to as errors.

Using (4.13), we can write

$$\frac{\partial a_j}{\partial w_{ji}} = z_i. \quad (4.17)$$

Substituting (4.16) and (4.17) into (4.15), we obtain

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i \quad (4.18)$$

Thus the derivative is obtained simply by multiplying the value of δ for the unit at the output end of the weight by the value z for the unit at the input end of the weight.

We only need to calculate the value δ_j for each hidden unit and output unit in the network and then apply (4.18).

For the output unit, we have

$$\delta_k = y_k - t_k \quad (4.19)$$

provided we are using the canonical link as output-unit activation function.

To evaluate the δ 's for hidden units, we again make use of the chain rule for partial derivatives,

$$\delta_j := \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j} \quad (4.20)$$

where the sum runs over all units k to which unit j sends connections. In the above equation we are making use of the fact that variations in a_j give rise to variations in the error function only through variations in a_k .

We obtain the backpropagation formula by substitution of the definition of δ , given by (4.16) into (4.20), and making use of (4.13) and (4.14)

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k \quad (4.21)$$

which tells us that the value of δ for a particular hidden unit can be obtained by propagating the δ 's backwards from units higher up in the network.

The backpropagation procedure can be summarized as follows.

Error backpropagation

1. Apply an input vector x_n to the network and forward propagate through the network using (4.13) and (4.14) to find the activations of all the hidden units and output units.
2. Evaluate the δ_k for all output units using (4.19).
3. Backpropagate the δ 's using (4.21) to obtain δ_j for each hidden unit in the network.
4. Use (4.18) to evaluate the required derivatives.

For batch methods like the HF method the total error E can then be obtained by repeating the above steps for each pattern in the training set and then summing over all patterns:

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E_n}{\partial w_{ji}} \quad (4.22)$$

Backpropagation is able to compute the gradient ∇E in $O(W)$, where W is the number of all weights in the network. For example the use of a finite difference approximation will need $O(W^2)$.

4.5. Fast multiplication by the Hessian

The computation of the Hessian of a neural network can be expensive if the number of weights W is large. In the context of optimization the Hessian is needed to compute the Newton direction $d^k = -\mathbf{H}_k^{-1} \nabla f_k$ in outer iteration k .

Truncated Newton methods are satisfied with an approximate solution of the Newton equations $\mathbf{H}_k d^k = -\nabla f_k$.

This solution can be found applying a Krylov subspace method like the Conjugate Gradient method with a suitable stopping criterion.

Therefore the Hessian is not needed, instead it suffices to be able to compute the matrix-vector product $\mathbf{H}\mathbf{v}$ for some arbitrary vectors \mathbf{v} which are needed in the CG iterations. Pearlmutter [30] showed that the computation of these matrix vector products can be computed efficiently in $O(W)$ using a variant of the backpropagation algorithm.

We first note that

$$v^T \mathbf{H} = v^T \nabla (\nabla E) \quad (4.23)$$

where ∇ denotes the gradient operator in weight space. We can write down the standard forward-propagation and backpropagation equations for the evaluation of ∇E and apply (4.23) to these equations to give a set of forward-propagation and backpropagation equations for the evaluation of $v^T \mathbf{H}$. This corresponds to acting on the original equations with a differential operator $v^T \nabla$. We follow Pearlmutter [30] and define the operator $\mathcal{R}\{\cdot\}$ to denote the operator $v^T \nabla$.

The analysis makes use of the usual rules of differential calculus, together with the result

$$\mathcal{R}\{w\} = \mathbf{v}. \quad (4.24)$$

We will only consider a simple two-layer network with output units having linear activation functions, so that $y_k = a_k$ together with a sum-of-squares error.

We consider the contribution of the error function for one pattern in the data set. The required vector is then obtained by summing over the contributions from each of the patterns separately.

The forward equations are given by

$$a_j = \sum_i w_{ji} x_i \quad (4.25)$$

$$z_j = h(a_j) \quad (4.26)$$

$$y_k = \sum_j w_{kj} z_j \quad (4.27)$$

Applying the $\mathcal{R}\{\cdot\}$ operator to obtain a set of forward propagation equations in the form

$$\mathcal{R}\{a_j\} = \sum_i v_{ji} x_i \quad (4.28)$$

$$\mathcal{R}\{z_j\} = h'(a_j) \mathcal{R}\{a_j\} \quad (4.29)$$

$$\mathcal{R}\{y_k\} = \sum_j w_{kj} \mathcal{R}\{z_j\} + \sum_j v_{kj} z_j \quad (4.30)$$

where v_{ji} is the element of the vector v that corresponds to the weight w_{ji} .

Because we are considering a sum-of-squares error function, we have the following

standard backpropagation expressions:

$$\delta_k = y_k - t_k \quad (4.31)$$

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k. \quad (4.32)$$

Applying the $\mathcal{R}\{\cdot\}$ operator leads to a set of backpropagation equations in the form

$$\mathcal{R}\{\delta_k\} = \mathcal{R}\{y_k\} \quad (4.33)$$

$$\mathcal{R}\{\delta_j\} = h''(a_j) \mathcal{R}\{a_j\} \sum_k w_{kj} \delta_k + h'(a_j) \sum_k v_{kj} \delta_k + h'(a_j) \sum_k w_{kj} \mathcal{R}\{\delta_k\}. \quad (4.34)$$

Finally we have the usual equations for the first derivatives of the error

$$\frac{\partial E}{\partial w_{kj}} = \delta_k z_j \quad (4.35)$$

$$\frac{\partial E}{\partial w_{ji}} = \delta_j x_i \quad (4.36)$$

and applying $\mathcal{R}\{\cdot\}$ we obtain expressions for the elements of the vector $v^T \mathbf{H}$

$$\mathcal{R}\left\{\frac{\partial E}{\partial w_{kj}}\right\} = \mathcal{R}\{\delta_k\} z_j + \delta_k \mathcal{R}\{z_j\} \quad (4.37)$$

$$\mathcal{R}\left\{\frac{\partial E}{\partial w_{ji}}\right\} = x_i \mathcal{R}\{\delta_j\}. \quad (4.38)$$

If desired, the technique can be used to evaluate the full Hessian matrix by choosing the vector v to be given successively by a series of unit vectors, each of which picks out one column of the Hessian.

4.6. Hessian and Gauss-Newton Hessian for neural networks

Following [9], the objective function f is defined as follows. For a given example x_p the j -th output unit is denoted $o_j(x_p, w)$. These outputs are then combined to produce a loss. For instance, in our auto-encoder task, the sum-of-squared error can be written as $\sum_j \varphi_j(o_j, x_p)$, with $\varphi_j(o_j, x_p) = \frac{1}{2} \|o_j(x_p, w) - x_p^j\|^2$. Commonly used loss functions can be written in this form with φ_j a convex function.

If the objective function f is of the form

$$f(w) = \sum_p \sum_j \varphi_j(o_j(x_p, w), x_p), \quad (4.39)$$

then the Gauss-Newton matrix \mathbf{H}_{GN} is defined as:

$$\mathbf{H}_{GN} = \sum_p J^p H^p (J^p)^T, \text{ with } J_{ij}^p = \frac{\partial o_j(x_p, w)}{\partial w_i} \text{ and } H_{jj}^p = \frac{\partial^2 \varphi_j(o_j(x_p, w), x_p)}{\partial o^2} \quad (4.40)$$

For simplicity, we have a closer look at a multilayer perceptron with a single output neuron. Following [17], the network is trained by minimizing the cost function

$$E_{av}(\mathbf{w}) = \frac{1}{2N} \sum_{i=1}^N [d(i) - F(x(i); \mathbf{w})]^2 \quad (4.41)$$

where $\{x(i), d(i)\}_{i=1}^N$ is the training sample and $F(x(i); \mathbf{w})$ is the approximating function realized by the network; the weights of the network are arranged in some orderly manner to form the weight vector \mathbf{w} . The gradient and the Hessian of the cost function $E_{av}(\mathbf{w})$ are respectively defined by

$$g(\mathbf{w}) = \frac{\partial E_{av}(\mathbf{w})}{\partial \mathbf{w}} \quad (4.42)$$

$$= -\frac{1}{N} \sum_{i=1}^N [d(i) - F(x(i); \mathbf{w})] \frac{\partial F(x(i); \mathbf{w})}{\partial \mathbf{w}} \quad (4.43)$$

and

$$\mathbf{H}(\mathbf{w}) = \frac{\partial^2 E_{av}(\mathbf{w})}{\partial \mathbf{w}^2} = \frac{1}{N} \sum_{i=1}^N \left[\frac{\partial F(x(i); \mathbf{w})}{\partial \mathbf{w}} \right] \left[\frac{\partial F(x(i); \mathbf{w})}{\partial \mathbf{w}} \right]^T \quad (4.44)$$

$$- \frac{1}{N} \sum_{i=1}^N [d(i) - F(x(i); \mathbf{w})] \frac{\partial^2 F(x(i); \mathbf{w})}{\partial \mathbf{w}^2} \quad (4.45)$$

To get the Gauss-Newton approximation of the Hessian, we ignore the second term on the right-hand side.

$$\mathbf{H}(\mathbf{w}) \approx \frac{1}{N} \sum_{i=1}^N \left[\frac{\partial F(x(i); \mathbf{w})}{\partial \mathbf{w}} \right] \left[\frac{\partial F(x(i); \mathbf{w})}{\partial \mathbf{w}} \right]^T \quad (4.46)$$

This approximation consists of the outer product of the partial derivative $\frac{\partial F(x(i); \mathbf{w})}{\partial \mathbf{w}}$ with itself, averaged over the training sample; accordingly, it is referred to as outer-product or Gauss-Newton approximation of the Hessian. The use of this approximation is justified when the algorithm is operating in the neighborhood of a local or global minimum. Far away from a minimum the Levenberg-Marquardt parameter λ will prevent us to put too much emphasis on \mathbf{H}_{GN} , thus the accuracy of the approximation plays only a minor role.

Schraudolph [31] showed how matrix-vector products of the Gauss-Newton Hessian can be computed efficiently using forward and backward passes through the neural network.

Part III.

Construction of the SPAI Preconditioner

Constructing a SPAI Preconditioner for the inner iteration of the HFO Method

It should be convenient to divide the HF algorithm into two phases related to the magnitude of the Levenberg parameter in $\mathbf{H}_{LM}^k = \mathbf{H}_{GN}^k + \lambda \mathbf{I}$.

In Phase I because of $\lambda \gg 0$ the diagonal elements of \mathbf{H}_{LM}^k will probably be the largest elements. Thus a diagonal preconditioner like the one Martens uses should be beneficial.

As the optimization proceeds the damping parameter λ gets smaller.

If $\lambda \approx 0$ the resulting system to be solved by the CG algorithm is the Newton equation $\mathbf{H}_{GN}d = -\nabla f$, where \mathbf{H}_{GN} is the Gauss-Newton approximation of the Hessian \mathbf{H} .

This is considered as Phase II.

In this phase the Hessian is approximately as dense as in Phase I but also more shallow considering the magnitude of the elements.

That's why the diagonal preconditioner used in Phase I is not as beneficial as in the earlier stages to speedup the CG algorithm. Thus using a preconditioner better suited for the solution of a dense linear system is clearly preferable.

First, we show how to obtain the needed a priori sparsity pattern for the SPAI preconditioner for a given matrix.

Then Section 5 discusses how some easily available information about the magnitude of the diagonal entries of the Gauss-Newton Hessian can be used to find a good sparsity pattern for $M = LL^T$ without the explicit computation of \mathbf{H} .

Preconditioning of the CG-step

Benzi showed in [5] that a SPAI preconditioner can be used successfully to speedup GMRES even for dense matrices. This is also true for other Krylov Subspace methods like CG.

Because a SPAI preconditioner needs an a priori pattern for the inverse of A , different approaches to determine a good pattern are discussed. The simplest heuristic to sparsify A is based on a threshold parameter. We use this approach to get some insight into the pattern of \mathbf{H} , and thus find a good pattern of \mathbf{H}^{-1} , at different stages of the optimization process. The Hessian is constructed column-wise, e.g. by constructing the j th column of G by computation of the matrix-vector product $\mathbf{H}_{GN}e_j$ where e_j is the j th unit vector.

5. Efficient pattern finding

In his implementation of the Hessian-free optimization algorithm Martens uses the following preconditioner:

$$M = \left[\text{diag} \left(\sum_{i=1}^D \frac{\partial f_i}{\partial \mathbf{w}} \odot \frac{\partial f_i}{\partial \mathbf{w}} \right) + \lambda I \right]^\alpha \quad (5.1)$$

The term $\sum_{i=1}^D \frac{\partial f_i}{\partial \mathbf{w}} \odot \frac{\partial f_i}{\partial \mathbf{w}}$ are the diagonal entries of the empirical Fisher matrix, thus they can be regarded as an approximation of the diagonal elements of \mathbf{H}_{GN} in the general case.

Schraudolph [31] defines the Fisher information matrix $F_{\mathcal{F}}$ of a scalar log-likelihood function $\mathcal{F}: \mathbb{R}^n \rightarrow \mathbb{R}$ as the $n \times n$ matrix formed by the outer product of its first derivatives.

$$(F_{\mathcal{F}})_{ij} = \frac{\partial \mathcal{F}(\mathbf{w})}{\partial \mathbf{w}_i} \cdot \frac{\partial \mathcal{F}(\mathbf{w})}{\partial \mathbf{w}_j} \quad (5.2)$$

$F_{\mathcal{F}}$ always has rank one. For more than one sample the Fisher information matrix proper is given by the expectation over all training samples, $\hat{F} = \langle F_{\mathcal{F}} \rangle_x$. It describes the geometric structure of weight space. For systems with a single linear output and sum-squared error, \mathbf{H}_{GN} reduces to \hat{F} . See Schraudolph [31] for more information on the differences between Fisher information and Gauss-Newton approximation.

Therefore Martens already computes a vector containing the approximation of the diagonal elements of the Hessian in a vector v .

$$v = \begin{pmatrix} \sum_i \left(\frac{\partial f_i}{\partial \mathbf{w}_1} \right)^2 \\ \vdots \\ \sum_i \left(\frac{\partial f_i}{\partial \mathbf{w}_i} \right)^2 \\ \vdots \\ \sum_i \left(\frac{\partial f_i}{\partial \mathbf{w}_n} \right)^2 \end{pmatrix}$$

Because of the particular pattern of the Gauss-Newton approximation of the Hessian we can use this vector to determine an efficient sparsity pattern of our preconditioner.

5.1. Gauss-Newton approximation

We follow Section 4.6 and first restrict ourselves to the case of only one output neuron. Then the Gauss-Newton Hessian is given by

$$\mathbf{H}(\mathbf{w}) \approx \frac{1}{N} \sum_{i=1}^N \left[\frac{\partial F(x(i); \mathbf{w})}{\partial \mathbf{w}} \right] \left[\frac{\partial F(x(i); \mathbf{w})}{\partial \mathbf{w}} \right]^T. \quad (5.3)$$

In more detail:

$$\begin{aligned} \frac{\partial F_i}{\partial \mathbf{w}} &= \begin{pmatrix} \frac{\partial F_i}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial F_i}{\partial \mathbf{w}_n} \end{pmatrix}, \quad \frac{\partial F_i}{\partial \mathbf{w}} \left(\frac{\partial F_i}{\partial \mathbf{w}} \right)^T = \begin{pmatrix} \frac{\partial F_i}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial F_i}{\partial \mathbf{w}_n} \end{pmatrix} \begin{pmatrix} \frac{\partial F_i}{\partial \mathbf{w}_1} & \dots & \frac{\partial F_i}{\partial \mathbf{w}_n} \end{pmatrix} \\ \left(\sum_i \frac{\partial F_i}{\partial \mathbf{w}} \left(\frac{\partial F_i}{\partial \mathbf{w}} \right)^T \right)_{kl} &= \sum_i \frac{\partial F_i}{\partial \mathbf{w}_k} \frac{\partial F_i}{\partial \mathbf{w}_l} \end{aligned} \quad (5.4)$$

Note that in the more general case with p outputs, the gradient $\frac{\partial F(x(i); \mathbf{w})}{\partial \mathbf{w}}$ has to be substituted by the Jacobian of the p outputs. See equation (4.40) for more details. Therefore the elements $\sum_i \frac{\partial F_i}{\partial \mathbf{w}_k} \cdot \frac{\partial F_i}{\partial \mathbf{w}_l}$ are only approximations of the real diagonal values. Nevertheless Figure 5.1 shows that using these entries already suffices to speed up GMRES.

5.2. Building the sparsity pattern

Note that in order to obtain a well-defined preconditioner $M = LL^T$, we have to ensure that the diagonal entries of M are nonzero.

For example consider the N -dimensional case, defining $l_j = \sum_i \left(\frac{\partial F_i}{\partial \mathbf{w}_j} \right)^2$, we get

$$w = \begin{pmatrix} l_1 \\ \vdots \\ l_N \end{pmatrix}.$$

Now we can sort this vector and only consider the products of the k largest elements. The products of these elements will have the largest values in the matrix and thus should be the first candidates to consider when looking for a good sparsity pattern for a preconditioner.

$$\bar{w} = \text{sort}(w) = \begin{pmatrix} l_i \\ l_l \\ \vdots \\ l_m \\ \vdots \\ l_j \end{pmatrix} \xrightarrow{l_m \text{ at position } k} \text{mark positions } (i, i), (l, l), (m, m), \dots, (i, l), (i, m) \text{ and } (l, m).$$

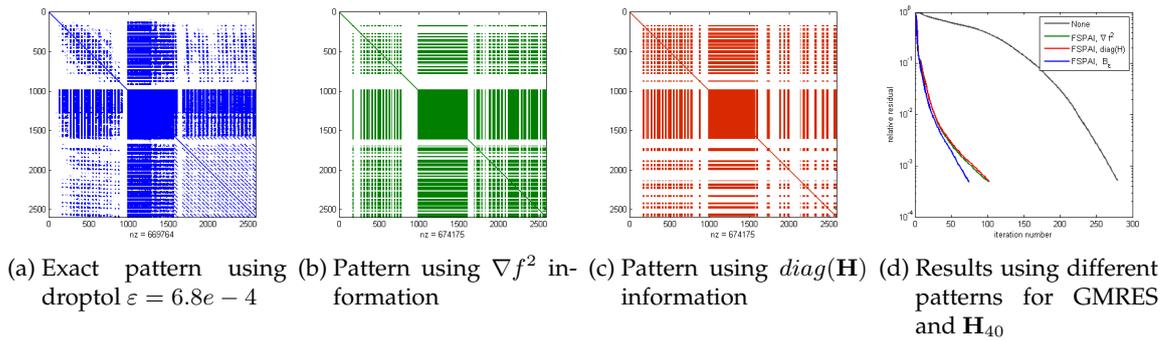


Figure 5.1.: Exact pattern and approximations using the patternfinding idea

To get a well-defined preconditioner the diagonal entries have to be nonzero. It is convenient to add the identity matrix.

5.3. Summary of the pattern finding method

- Set k to define the allowed sparsity of the pattern. k is the cutoff number of elements in v to consider for the pattern.
It is directly related to the sparsity of the pattern by $k = \sqrt{(1-\text{Sparsity}\%) * \text{Problemsize}^2}$.
- Compute squared gradient as approximation of the diagonal of the Hessian. Because of the particular pattern of the Gauss-Newton approximation the products of the largest elements in this vector will tend to be the largest elements in \mathbf{H}_{GN} .
- Sort v and save the the k -th largest element.
- Find all elements in v that are larger than $v(k)$.
- Find all positions where two of the k largest elements will be multiplied.
- Mark these positions in a matrix C and add the identity matrix to ensure that the preconditioner is well-defined.

Figure 5.1a to 5.1c show the exact pattern of \mathbf{H}_{GN} using a predefined drop tolerance ε and the approximate pattern obtained using the pattern finding algorithm mentioned above. The results using these patterns as preconditioners for GMRES are shown in Figure 5.1d. If the pattern finding method is used, the preconditioner reduces the needed iterations from 300 to approximately 110. The use of the exact diagonal allows GMRES to finish after only 80 iterations.

It is remarkable that the use of the pattern finding approach gives almost as good results

as the use of the sparsified matrix $\mathbf{H}_{GN,\varepsilon}$. Listing 5.1 shows the Matlab implementation of this method.

```
function C = patternFinding(v,k,psize)
% v      vector containing an approximation of the diagonal of the
%        Gauss-Newton Hessian
% k      number of largest elements that should be considered
%        It is directly related to the sparsity of the preconditioner
%        by  $k = \sqrt{(1-\text{Sparsity\%}) * \text{Problemsize}^2}$ 
% psize  problemsize, in our testcase psize = 2595

[w,idx] = sort(v,'descend');
r = w(k);
t = v >= r;
% convert logical to double
t = sparse(+t);
% add diagonal matrix to ensure that the preconditioner is well-defined
C = t*t' + speye(psize,psize);
```

Listing 5.1: The pattern finding algorithm in Matlab

Part IV.

Case study

6. Training of a deep auto-encoder

6.1. Overview

The main purpose of this thesis is to apply a SPAI preconditioner in the context of optimization using a truncated newton method by preconditioning the Krylov subspace method used in the inner iteration of the nonlinear optimization. It doesn't matter if GMRES or CG is used.

Our test-case is the training of a deep auto-encoder using the HF algorithm as it is done by Martens [26].

As dataset we use the CURVES dataset, which is an artificial dataset consisting of grayscale images of curves at 28×28 resolution. It consists of 20K training samples and 10K testing samples. Random samples from the dataset are shown in Figure 6.2a ¹. Auto-encoders are used for dimensionality reduction.

See [18] for a comparison of different methods. It is trained with the objective to minimize the reconstruction error. The activations of the smallest layer then yield an encoded low-dimensional representation of the input. Martens code is training a deep auto-encoder using an 784-400-200-100-50-25-6 architecture for the encoding part of the neural network.

This leads to about 900k weights that have to be training. The learning is done in parallel on a GPU using Jacket².

Because of the unavailability of a Jacket license, we consider a simplified problem.

We only consider every sixteenth pixel from the input, reducing the dimensionality of the input from 784 to 49. Then we train a deep auto-encoder with a 49-20-10-6 encoding layer. This leads to $N = 2595$ weights that have to be trained.

The structure of the problem remains the same even though the input space is reduced. Therefore the results should also be applicable for the higher dimensional case using GPUs.

Through the reduction of the input space, we are able to compute the Hessian or more specific the Gauss-Newton Hessian for some fixed HF iterations. We run the HF algorithm for 50 iterations.

¹Figure copied from [18]

²<http://www.accelereyes.com/>

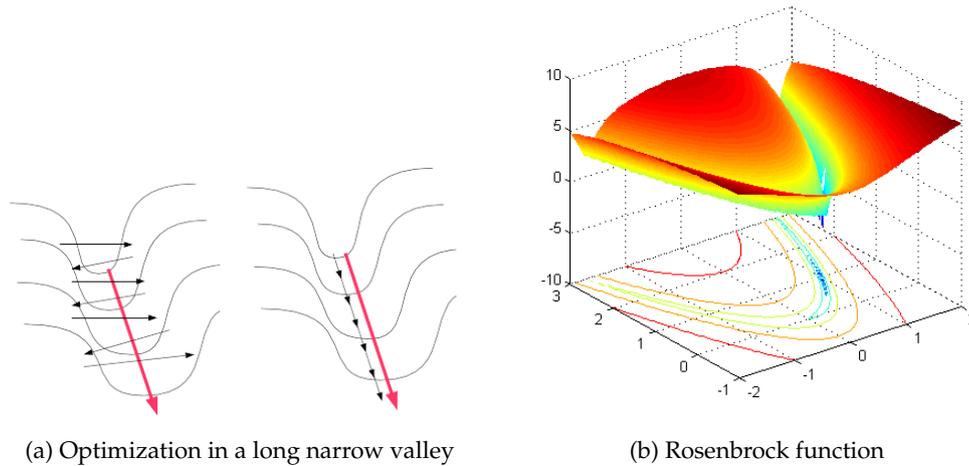


Figure 6.1.: The Rosenbrock function as example of pathological curvature

6.2. Pathological Curvature

Before we start our analysis of the problem, it is useful to motivate the use of curvature information in the optimization process. Following Martens [25], the problems associated with the learning of deep neural networks like deep auto-encoder can be explained by regions of pathological curvature in the objective function which resemble bad local minima to 1st-order optimization methods.

By taking curvature into account, Newton's method rescales the gradient, so it is a much more sensible region to follow.

If the curvature is low and positive in a particular descent direction d , the gradient changes slowly along d . Thus d will remain a descent direction over a long distance. Therefore we should choose a search direction p that travels far along d , even if the amount of reduction associated with d , $\nabla f^T d$ is relatively small.

If the curvature is high, then we should choose p such that the distance traveled along d is smaller.

Newton's method computes the distance to move along d as its reduction divided by its curvature: $-\nabla f^T d / d^T H d$. Not taking curvature information into account for the computation of the search direction can lead to undesirable scenarios:

- The sequence of search directions might constantly move to far in directions of high curvature, causing the typical zig-zag path that is associated with common gradient descent methods. This can be circumvented by decreasing the learning rate.
- Directions of low curvature will be explored much more slowly than they should be. Reducing the learning rate will make things worse.
- If only directions of significant decrease in f are ones of low curvature, then the

optimization may become too slow and can appear to be trapped in a local minimum.

Figure 6.1a³ shows the optimization of an objective that locally resembles a long narrow valley. At the base of the valley is a direction of low curvature and reduction that needs to be followed.

The smaller black arrows show the steps taken by gradient descend with large and small learning rate. The red arrows are the step computed by Newton's method.

The case is pathological because of the mixture of low and high curvature directions.

Figure 6.1b shows the Rosenbrock function as an example of a function that exhibits pathological curvature. A long narrow valley has to be followed to find the minimum at $(0, 0)$.

6.3. Analysis

We compute the Gauss-Newton Hessian explicitly through the computation of matrix vector products with unit vectors e_j , $j = 1, \dots, N$.

SPAI preconditioners are usually applied to sparse matrices.

Note that the Hessian is dense. In our testcase about 85% of the elements are nonzero throughout the optimization.

In Figure 6.2b the pattern of the Hessian in our reduced testcase is shown.

To get some insight into the structure of the Hessian, we save it for iteration 10, 20, 30 and 40.

We refer to these matrices as \mathbf{H}_{10} , \mathbf{H}_{20} , \mathbf{H}_{30} and \mathbf{H}_{40}

Following Benzi [5], we first normalize the Hessians. It is self-evident to expect to find the largest elements of \mathbf{H}^{-1} at the same positions as \mathbf{H} . To come up with a sparsity pattern for our preconditioner M , the simplest criterion is to choose a threshold parameter $\varepsilon \in (0, 1)$ and to include position (i, j) in the nonzero pattern of M if

$$|a_{ij}| > \max_{1 \leq k, l \leq n} |a_{kl}|. \quad (6.1)$$

In Figure 6.3 the sparse matrices $\mathbf{H}_{10, \varepsilon}$ to $\mathbf{H}_{40, \varepsilon}$ obtained for the different matrices \mathbf{H}_{10} to \mathbf{H}_{40} are shown.

Note that the identity matrix is added to every pattern to produce a feasible sparsity pattern for the SPAI preconditioner.

Even though at first glance it seems like all four pattern have a different sparsity, the sparsity of the preconditioners is at least 99%.

These matrices are used to precondition GMRES for the solution of $\mathbf{H}_i x = b$ with $b = \text{ones}(\text{psize})$.

³Copied from [25]

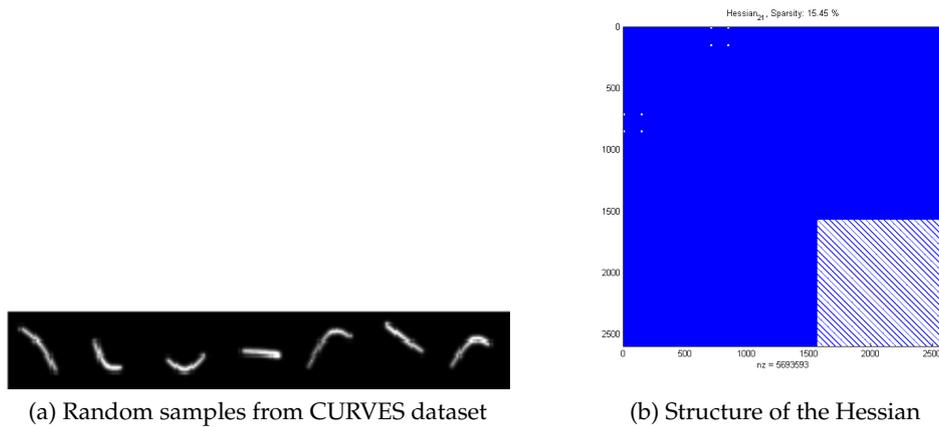


Figure 6.2.: Random samples of the CURVES dataset and dense pattern of the Hessian

Even though CG is used in the inner HF iteration, we use GMRES out of convenience to not have to care about the stopping criteria of CG and more important, GMRES can also handle unsymmetric preconditioners that were used in earlier tests.

We investigate the eigenvalue distributions for the different matrices before and after applying the SPAI preconditioner. Note that the Gauss-Newton Hessian is symmetric positive definite, therefore all eigenvalues are positive.

The eigenvalue distribution of the different unpreconditioned and preconditioned matrices are shown in Figure 6.4 to Figure 6.7. Especially for the matrices in the early optimization the factorized preconditioner L deflates most of the eigenvalues to 1.

These matrices were used in our GMRES testcase.

The exact values and the number of GMRES iterations needed for convergence with and without preconditioning are shown in Table 6.1. I corresponds to the unpreconditioned solution.

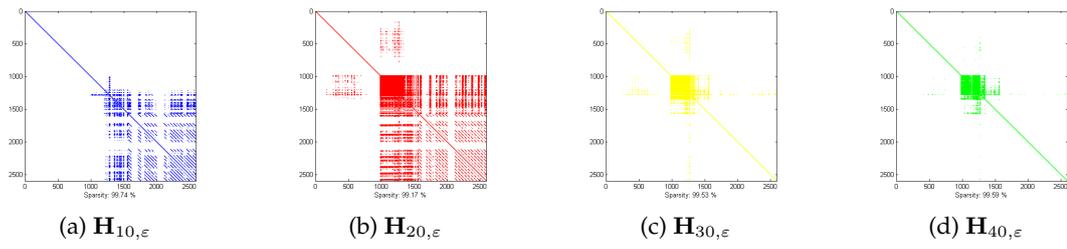
It is evident that the number of needed GMRES iterations increases as the optimization proceeds. This is mostly due to the following fact.

In the early stages of optimization, when x_k is far away from the optimum the quadratic approximation $q_\theta(p)$ is not reasonable.

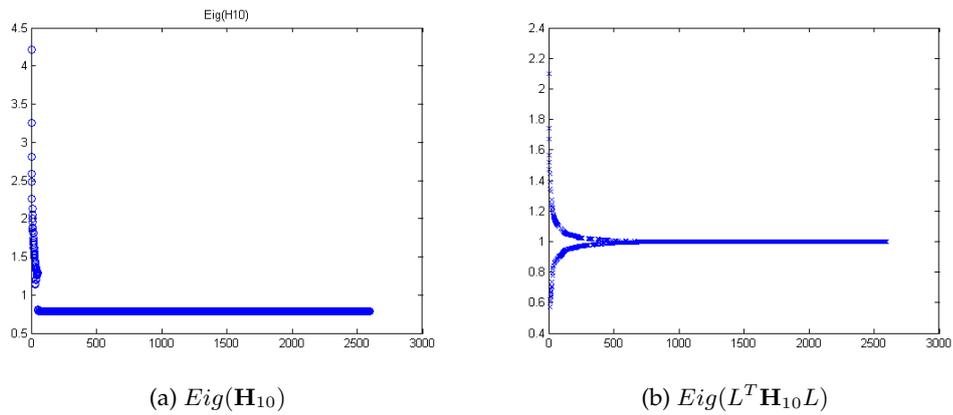
The Levenberg-Marquardt parameter λ in the modified Newton equation $(\mathbf{H} + \lambda I)d = -\nabla f$ controls how much emphasis is given on the information of the Hessian and thus the quadratic model in general.

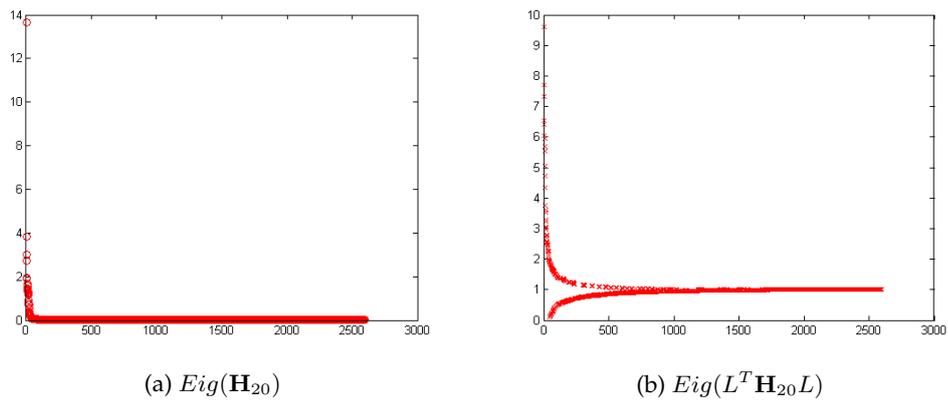
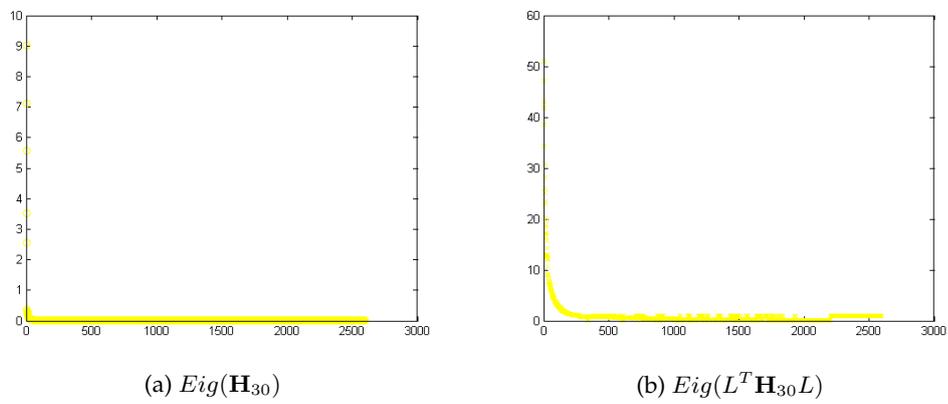
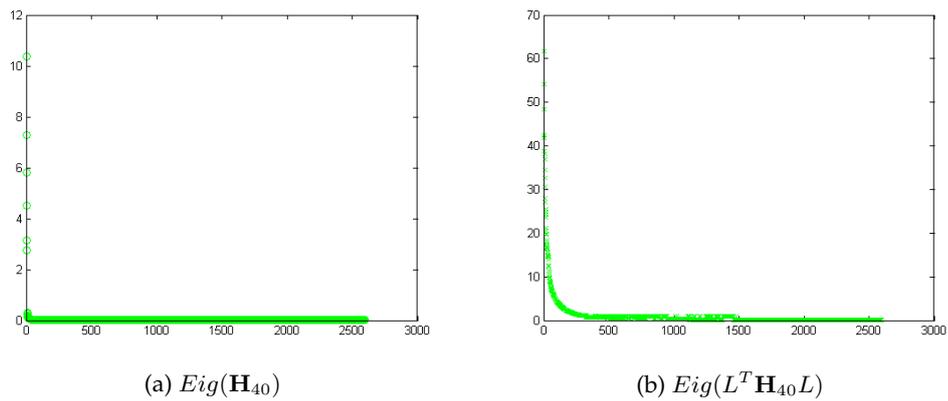
Far away from an optimum λ is chosen to be large and thus the matrix $\mathbf{H} + \lambda I$ will have its largest elements on the diagonal, leading to an easy to minimize function for a Krylov subspace method like GMRES.

But if x_k is near an optimum, λ will be almost zero and the Hessian won't have the largest elements in magnitude on the diagonal.

Figure 6.3.: Patterns for \mathbf{H}_{10} to \mathbf{H}_{40} using drop tolerance $\varepsilon = 0.025$

	I	$\mathbf{H}_{10,\varepsilon}$	I	$\mathbf{H}_{20,\varepsilon}$	I	$\mathbf{H}_{30,\varepsilon}$	I	$\mathbf{H}_{40,\varepsilon}$
GMRES	7	6	31	24	68	81	186	108
Sparsity	-	99.74%	-	99.17%	-	99.53%	-	99.59%

Table 6.1.: Results applying FSPAI preconditioner at different iterations of HF using drop tolerance $\varepsilon = 0.025$ Figure 6.4.: Eigenvalue distributions of \mathbf{H}_{10} and $L^T \mathbf{H}_{10} L$

Figure 6.5.: Eigenvalue distributions of \mathbf{H}_{10} and $L^T \mathbf{H}_{20} L$ Figure 6.6.: Eigenvalue distributions of \mathbf{H}_{30} and $L^T \mathbf{H}_{30} L$ Figure 6.7.: Eigenvalue distributions of \mathbf{H}_{40} and $L^T \mathbf{H}_{40} L$

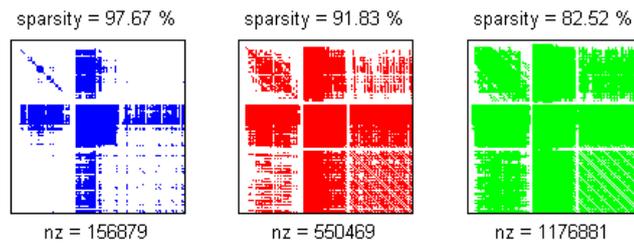


Figure 6.8.: Resulting pattern allowing different sparsities for \mathbf{H}_{40} .

The number of nonzero elements is from left to right: 2.3%, 8.1% and 17.5%

6.4. Results for different sparsity pattern

Now we investigate the resulting quality of our SPAI preconditioner in terms of GMRES iterations for patterns with decreasing sparsity.

In Figure 6.8 three different sparsity pattern for L_{40} are shown.

We use these patterns to precondition \mathbf{H}_{40} .

As we can see in Figure 6.9 the number of iterations decreases with decreasing sparsity of the chosen pattern.

We compare the performance of our preconditioner with other preconditioners.

In Figure 6.10 the performance of Jacobi, Gauss-Seidel and ILU is plotted for \mathbf{H}_{40} .

Figures for the other iterations can be found in the appendix.

As we can see the SPAI preconditioner outperforms both Jacobi and Gauss-Seidel preconditioner for these dense matrices.

Note that the ILU preconditioner is obtained using a tolerance of 10^{-5} for our normalized matrices. Therefore ILU basically corresponds to the computation of the exact inverse.

The quality of SPAI in terms of reduction of iterations can be regulated by the density of the underlying pattern.

The different pattern for Table 6.2 are shown in the appendix.

	\mathbf{H}_{10}	Sparsity	\mathbf{H}_{20}	Sparsity	\mathbf{H}_{30}	Sparsity	\mathbf{H}_{40}	Sparsity
I	7	-	31	-	64	-	97	-
P1	4	98.28%	10	95.60%	51	97.00%	76	97.00%
P2	3	93.63%	6	88.40%	22	90.00%	46	91.80%
P3	2	84.45%	4	78.00%	10	80.00%	18	82.50%

Table 6.2.: Results applying FSPAI preconditioner at different iterations of HF using different patterns

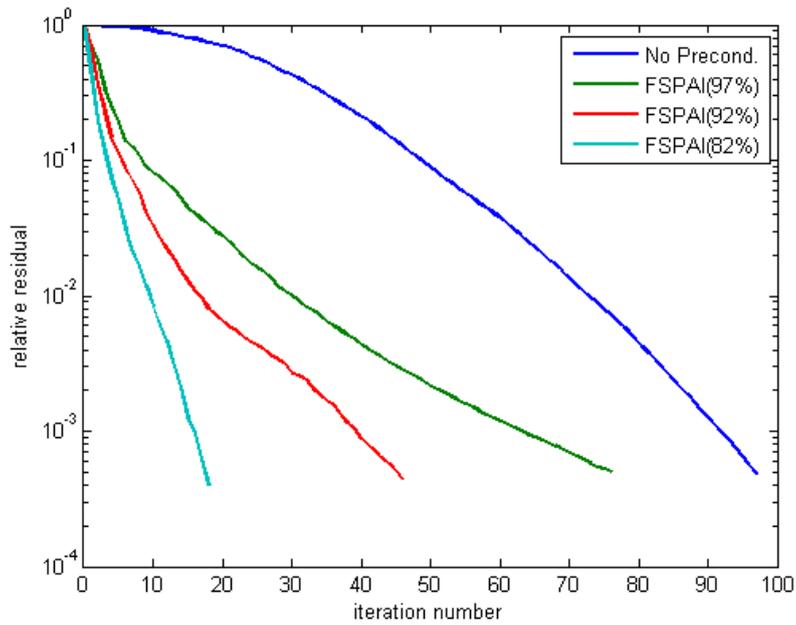
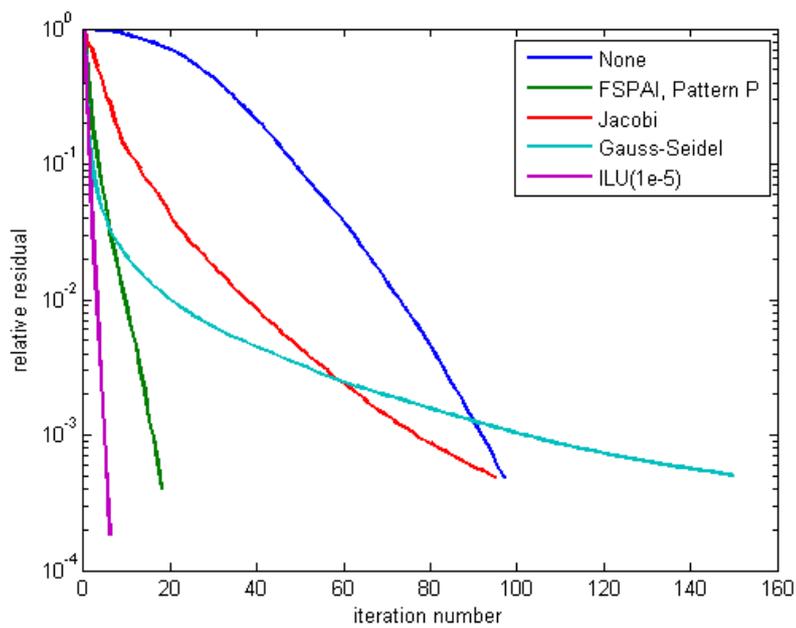


Figure 6.9.: FSPA for different sparsity patterns

Figure 6.10.: Different preconditioners for H_{40}

6.5. Results for reduced models

We have seen that a SPAI preconditioner with an appropriate sparsity pattern is able to heavily reduce the needed GMRES iterations at every iterate of the outer iteration of the Hessian-Free optimization method, we are able to apply it in the context of HF optimization.

Even though the computation of our preconditioner is embarrassingly parallel, it might not be economical to compute it at every outer iteration. Therefore we are testing different update strategies for our preconditioner.

In the first ten iterations of our testcase even the unpreconditioned inner iteration will converge fast because of the Levenberg-Marquardt parameter λ being large, see Figure 6.13b.

Thus we will only start to use our preconditioner if $\lambda \approx 0$. In the test-case we will start to apply our preconditioner at iteration 10, then use it for the next ten consecutive iterations and update it using a new pattern and use this for the next ten iterations.

Furthermore we use our pattern finding idea to compute our desired sparsity pattern using ∇f^2 information as described in Section 5. We will call this update strategy SPAI(10).

Note that we are using a more dense pattern the longer our optimization is running to compensate for the more shallow pattern of the Gauss-Newton Hessian.

We start with a sparsity of 90% and continually decrease the sparsity up to 80%.

As second strategy, we will update our preconditioner every five iterations. We call this update SPAI(5).

And to show how the preconditioner can speedup the later stages of optimization, we use the above update strategy, but when reaching iteration 40 we will update our preconditioner every iteration.

The results are plotted in Figure 6.11a and Figure 6.12a .

As we can see the application of the SPAI preconditioner reduces the needed iteration at the outer iteration where it was applied.

Furthermore the application of the same preconditioner for consecutive iterations also reduces the needed iterations.

Figure 6.12b shows the results, when we use the approximate patterns obtained by our pattern finding method.

The diagonal preconditioner of Martens only remains competitive by the strange reductions between iteration 40 and 50 yielding to a way better as expected performance. Figure 6.11b shows the reduction of the GMRES iteration if we disregard the costs of computing the preconditioner at every iterate after iteration 40.

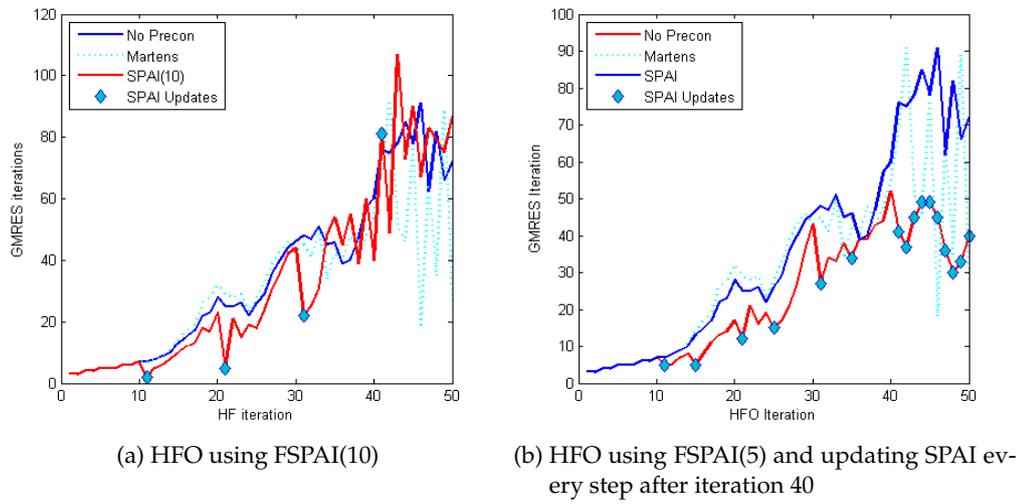


Figure 6.11.: HFO using different FSPAI update heuristics

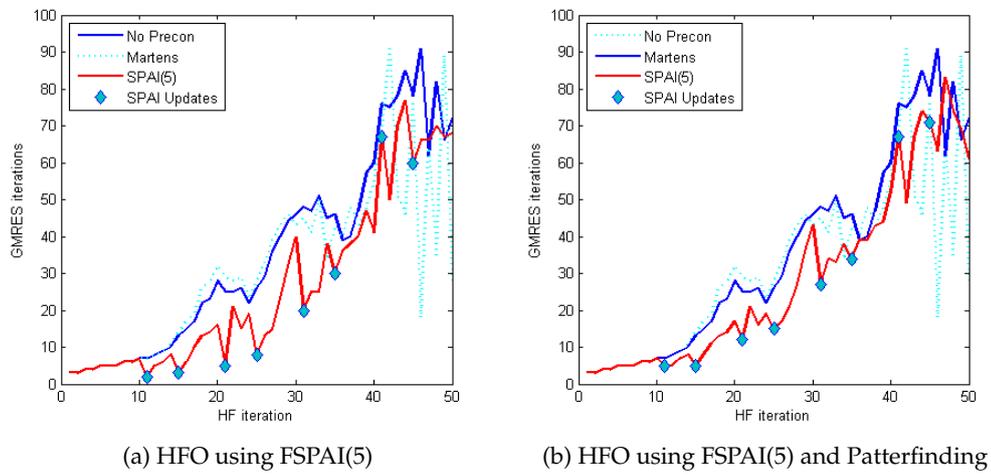


Figure 6.12.: Comparison of HFO using FSPAI(5) with exact pattern and approximate pattern

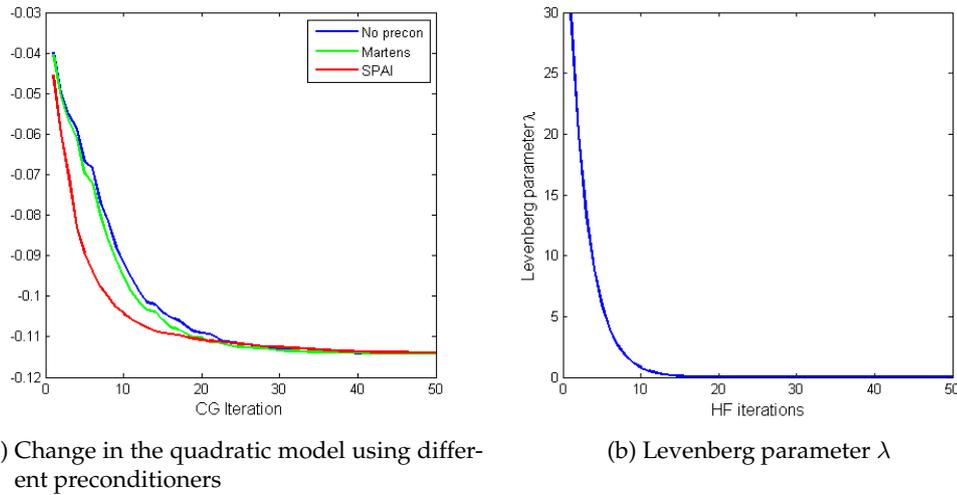


Figure 6.13.: Change in the quadratic model using different preconditioners and change of damping parameter λ

6.6. Results using CG for the inner iteration

The Gauss-Newton Hessian used in the HF algorithm is symmetric positive definite. Therefore the Conjugate Gradient method should be employed to solve the Newton equations. This is because on the one hand CG finds an optimal solution in the Krylov space $K^k(\mathbf{H}, \nabla f)$ like it is the case for GMRES, but on the other hand CG is able to cheaply construct a basis of K^k using two coupled two-term recurrences.

Martens uses a particular stopping criterion which proved useful.

If the stopping criterion is related to the minimization of the quadratic model then we can efficiently compute the value of the model in the CG iteration by $Q(p^{i+1}) = \frac{1}{2}(r_{i+1} + g)^T p^{i+1}$ as it is shown in Section 3.5.

Figure 6.13a shows the change in the quadratic model using different preconditioners in the latter stage of optimization, when no stopping criterion is applied. First note that all three quadratics have the same global minimum.

But, as expected, our preconditioner enables CG to get close to the minimum in only a few iterations. A stopping criterion based on the change in the quadratic model should be able to recognize this by stopping earlier, when a better preconditioner is used.

The performance of the different preconditioners varies with the choice of the stopping criterion of the CG algorithm.

Therefore we test different choices.

1. Residual-based stopping criterion: $\|r\| < \varepsilon$ or $N = N_{max}$
2. Martens stopping criterion based on the change in the quadratic:
 $Q(p^{i-1}) < 0$ and $(Q(p^i) - Q(p^{i-1}))/Q(p^i) < k\varepsilon$ or $N = N_{max}$

3. Combined residual- and quadratic-based stopping criteria, where the latter one is again Martens criterion.

We use $\varepsilon = 0.0005$ and $N_{max} = 500$ for all runs. Table 6.3 shows the log-likelihood on the testset, the testset error and the total number of CG iterations needed for the different runs.

Note that, in practice, the number of needed CG iterations is a good measure for the training time of the neural network. The generalization error, defined as the error of the network for an unknown test set, is the most important result. Our goal was to maximize the likelihood of our model, given the data.

Note that the stopping criteria based on the residual results in a huge number of CG iterations and gives the worst results on the test set. This shows that a residual-based stopping criteria is, in fact, a poor choice because of its divergent character.

Martens stopping criterion, which is based on the change in the quadratic model, leads to less CG iterations and better results on the test-set. This is probably because the stopping criterion forces CG to stop after only a few iterations in the beginning of the optimization, but in later stages it permits CG to explore the regions of pathological curvature that are related to small eigenvalues of the Hessian of the sum-of-squares error of the network.

The combined stopping criterion reduces the number of CG iterations but gives worse results on the test-set. This is probably because sometimes a small residual forces CG to stop even though there still can be made significant progress in the optimization of the quadratic model.

The best results are observed when using SPAI(10) and Martens stopping criterion, both, for the log-likelihood and the error on the test-case.

Stopping Criteria	Type	Test Log-likelihood	Testset Error	Total CG Iterations
Residual	SPAI(10)	-4,2103	0,1588	7314
	SPAI(5)	-4,2216	0,1583	7472
	I	-4,2094	0,1609	3149
	Martens	-4,2023	0,1526	4523
Quadratic	SPAI(10)	-4,0675	0,1281	2263
	SPAI(5)	-4,1133	0,1394	2193
	I	-4,1529	0,1487	2512
	Martens	-4,2120	0,1546	1639
Residual/Quadratic	SPAI(10)	-4,1516	0,1423	1940
	SPAI(5)	-4,1263	0,1370	2217
	I	-4,1357	0,1455	2271
	Martens	-4,1941	0,1542	1639

Table 6.3.: HF Results using CG in the inner iteration for different stopping criteria

Part V.

Conclusions and Future Work

7. Conclusions

There are many ways to improve optimization algorithms like truncated-newton methods. Other than the Newton method, truncated-Newton methods solve the Newton equations only approximately at every outer iteration using a Krylov subspace method. Usually the positive semi-definite Gauss-Newton Hessian is used. Thus the Conjugate Gradient algorithm is the preferred algorithm used in the inner iteration of the truncated-Newton method.

This thesis looked at the preconditioning of the inner iteration. Usually only diagonal preconditioners are considered.

We don't restrict ourselves to diagonal preconditioning.

Therefore we apply a sparse approximate inverse preconditioner (SPAI), although this type is commonly used for sparse matrices.

The most important task is to find an appropriate sparsity pattern. This is expensive considering that the Hessian is only given implicitly.

We use some cheaply available information about the magnitudes of the diagonal elements of the Hessian and the knowledge of how it is made up, to come up with an effective procedure to construct the needed sparsity pattern for a previously fixed number of nonzero elements.

Furthermore we consider the test-case of training a deep auto-encoder. We analyze the structure of the Hessian in different stages of the optimization by sparsification. Then we used these matrices as sparsity pattern for our SPAI preconditioner and showed the indirect relationship between the performance of our preconditioner and the sparsity of the underlying sparsity pattern.

Then we compared our preconditioner with other commonly used preconditioner like Jacobi, Gauss-Seidel and ILU.

We observe the superiority of our approach in terms of reductions of needed iterations of the GMRES method.

Then we've successfully applied our preconditioner to the training of the deep auto-encoder using different update strategies.

As expected the runtime of the Krylov method used in the inner iteration is significantly decreased at iterates where the SPAI preconditioner is applied. Furthermore even at consecutive iterates where the same preconditioner is used, the number of needed iterations is still reduced.

8. Future Work

8.1. Parallelization of the method

One of the advantages using a sparse approximate inverse preconditioner is that its computation is embarrassingly parallel. Because the learning of neural networks is already using the power of parallel computations on graphic cards, the actual computation of a SPAI preconditioner will be faster if a better graphics card with more potential of parallel computing is used. This can be a key aspect when considering to apply SPAI in the context of Hessian-Free optimization. In a typical SPAI implementation the preconditioner is computed columnwise. We need to perform W matrix-vector products with unit vectors to compute the columns of the Hessian.

If we perform this computation on one single processor the computation needs $O(W)$ operations to compute one column of the Hessian using forward and backward passes through the neural network. Because of the Hessian consisting of W columns the cost to compute our SPAI preconditioner needs $O(W^2)$ operations. This clearly prohibits its application on a single processor for deep networks.

Nevertheless the power of SPAI lies in the great performance increase on parallel architectures.

8.2. Termination of CG

There are a lot of different possibilities concerning the early termination of the CG method in the context of truncated-Newton methods used for nonlinear optimization. A good stopping criterion is based on the change in the quadratic model but it is also connected to a forcing sequence to guarantee fast convergence [12]. Suppose we have found a good stopping criterion, then the quality of our method will depend on the choice how many CG iterations we allow to be done when the criterion is not satisfied. This choice is problem dependent but it will have a huge influence in the overall performance of our truncated Newton method.

8.3. Application of a better approximation of the diagonal of the Hessian

In [9] Chappelle proposed a method to compute an unbiased estimate of the diagonal of the Hessian. It is shown that this Jacobi type preconditioner gives better results than

the preconditioner Martens used. Martens preconditioner uses a more crude approximation of the diagonal of the Hessian.

We are using these values to compute the sparsity pattern of our preconditioner. Of course the performance of our preconditioner will be increased, if we use a better approximation of the diagonal values. It would be interesting to see how the performance of the preconditioner varies using these approximations to the diagonal especially with decreasing sparsity.

Furthermore, we should use the diagonal values of the Gauss-Newton Hessian to improve the Levenberg-Marquardt damping, replacing $\mathbf{H}_{GN} + \lambda \mathbf{I}$ by $\mathbf{H}_{GN} + \lambda \text{diag}(\mathbf{H}_{GN})$.

8.4. Using more information for the preconditioner

In the process of optimization, the inner and outer loop contains information that can also be used to produce useful preconditioners.

Fassano uses information obtained while running CG to compute an appropriate preconditioner.

“The basic idea is to deflate the eigenvalues of the matrix of the system associated with the invariant subspace explored by the CG”(cf [15]).

8.5. Efficient calculation of particular entries of the Hessian

In its current implementation we need to compute every column of the Hessian to compute our preconditioner. As the computation of one column can be performed in $O(W)$, where W is the number of weights in the neural network, this leads to a complexity of $O(W^2)$ to actually compute the SPAI preconditioner. This clearly prohibits its application in a truncated-Newton method. This problem can be solved to some extent if the computation is performed in parallel on a graphic card.

Our preconditioner is usually sparse, e.g. 95% of the elements are zero.

In the columnwise computation of our preconditioner, we compute a column of the Hessian, which has size W . But we only need 5% of the entries of this usually large vector.

Bishop [7] showed how particular elements of the Hessian can be computed. Even though this method also needs $O(W^2)$ operations to compute the full Hessian, it seems possible to reduce the overall complexity of the computation of our preconditioner to $O(W)$ or $O(W \log(W))$.

An element of the Hessian is computed by a similar forward and backpropagation procedure as the gradient but some additional values have to be stored.

Our pattern finding algorithm, no matter if used with the diagonal of the Fischer matrix or the real values of the Hessian, supplies us with the positions of the needed elements in the Hessian. But if we know which elements we need to compute, then we only need to backpropagate through some part of the network. Thus we don't compute the 95% of information that we actually don't need for the computation of our preconditioner.

Furthermore we only have to calculate $\frac{1}{2}N(N + 1)$ elements, because of the symmetry of the Hessian and the factorized character of our preconditioner.

Therefore it seems reasonable to believe to come up with an algorithm that has a better complexity than $O(W^2)$.

Suppose we have an $O(W)$ algorithm, then our preconditioner is perfectly suited for the application in a truncated-Newton method that is used to train deep neural networks.

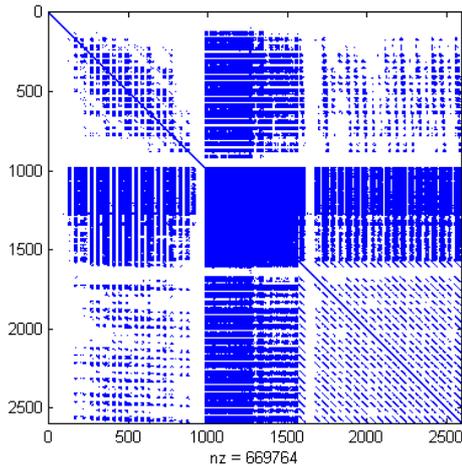
In his more recent paper Martens successfully applies his HF algorithm to train recurrent neural networks(RNNs) [26]. These networks are known to be really hard to train. He notes that his diagonal preconditioner is not working for this kind of problems. This is probably due to the fact mentioned in this thesis. Diagonal preconditioners seem to be not well suited if applied to dense matrices where a lot of elements are in the same order of magnitude.

Our preconditioner is designed to speed up the inner iteration of the HF optimization in this cases. This should prove extremely useful to speedup HF for the training of RNNs.

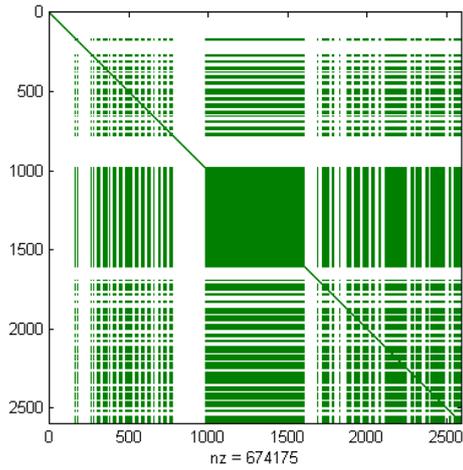
Of course, this is only true if we come up with a procedure to efficiently compute the needed entries of the Hessian.

Appendix

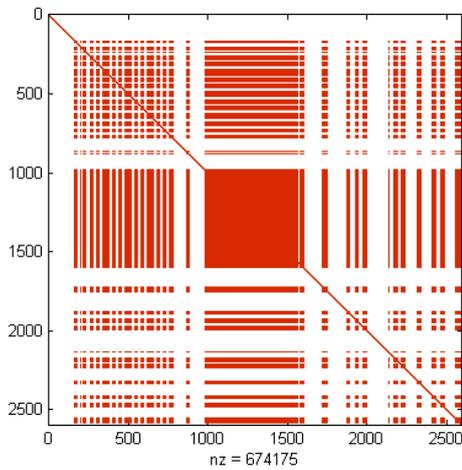
A. Pattern finding idea



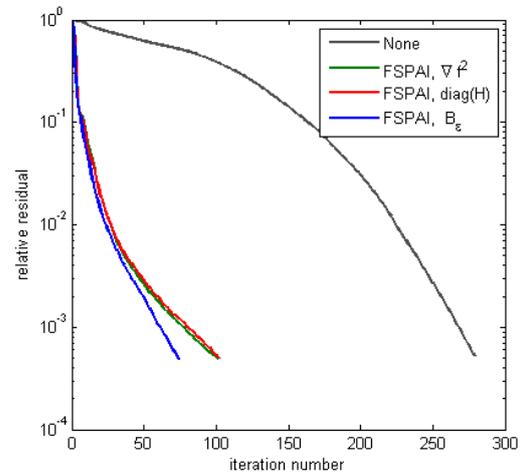
(a) Exact pattern using droptol $\epsilon = 6.8e-4$



(b) Pattern using ∇f^2 information



(c) Pattern using $\text{diag}(H)$ information



(d) Results using different patterns for GMRES and H_{40}

Figure A.1.: Exact pattern and approximations using the patternfinding idea

B. Sparsity Patterns

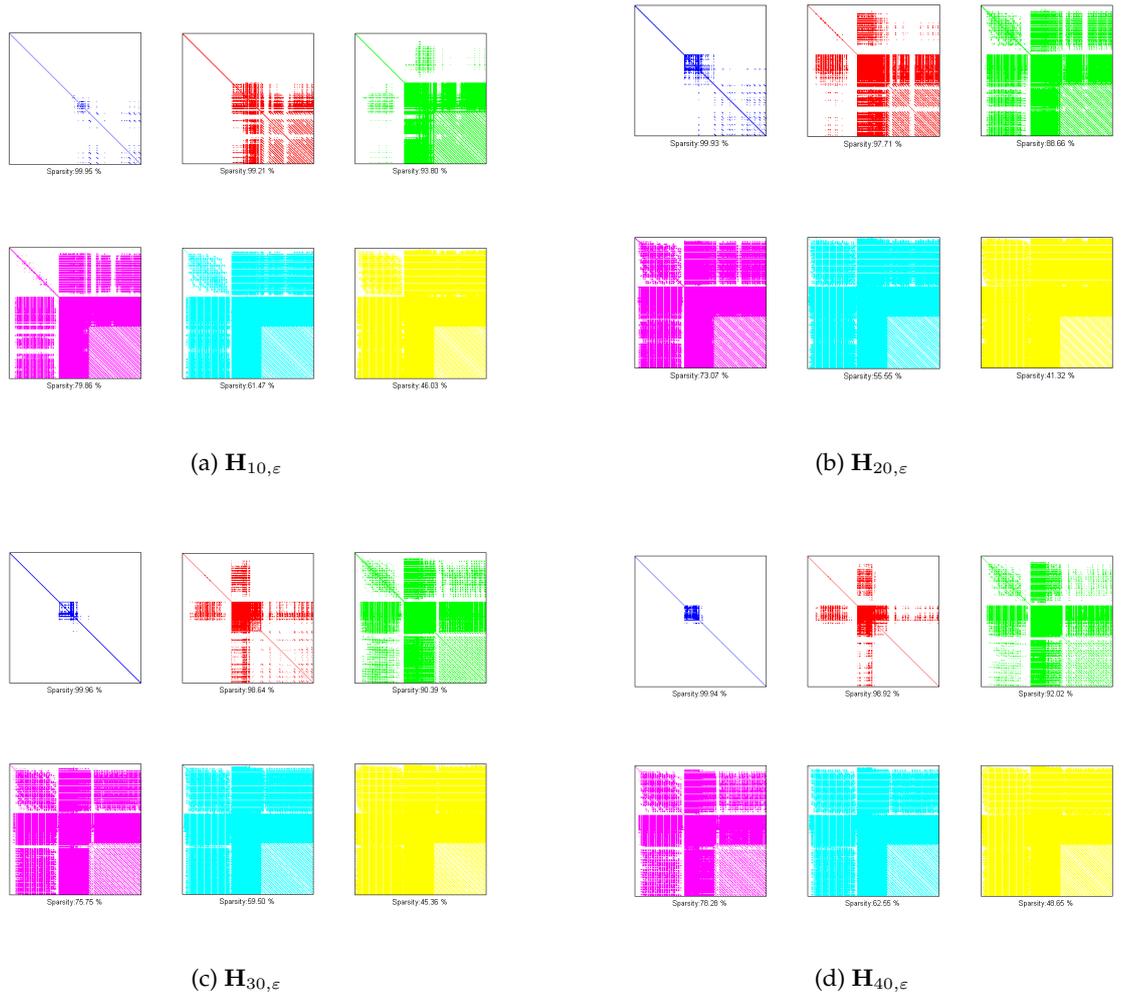


Figure B.1.: Sparsity patterns for matrices \mathbf{H}_{10} , \mathbf{H}_{20} , \mathbf{H}_{30} , \mathbf{H}_{40} for different drop tolerances ε_i

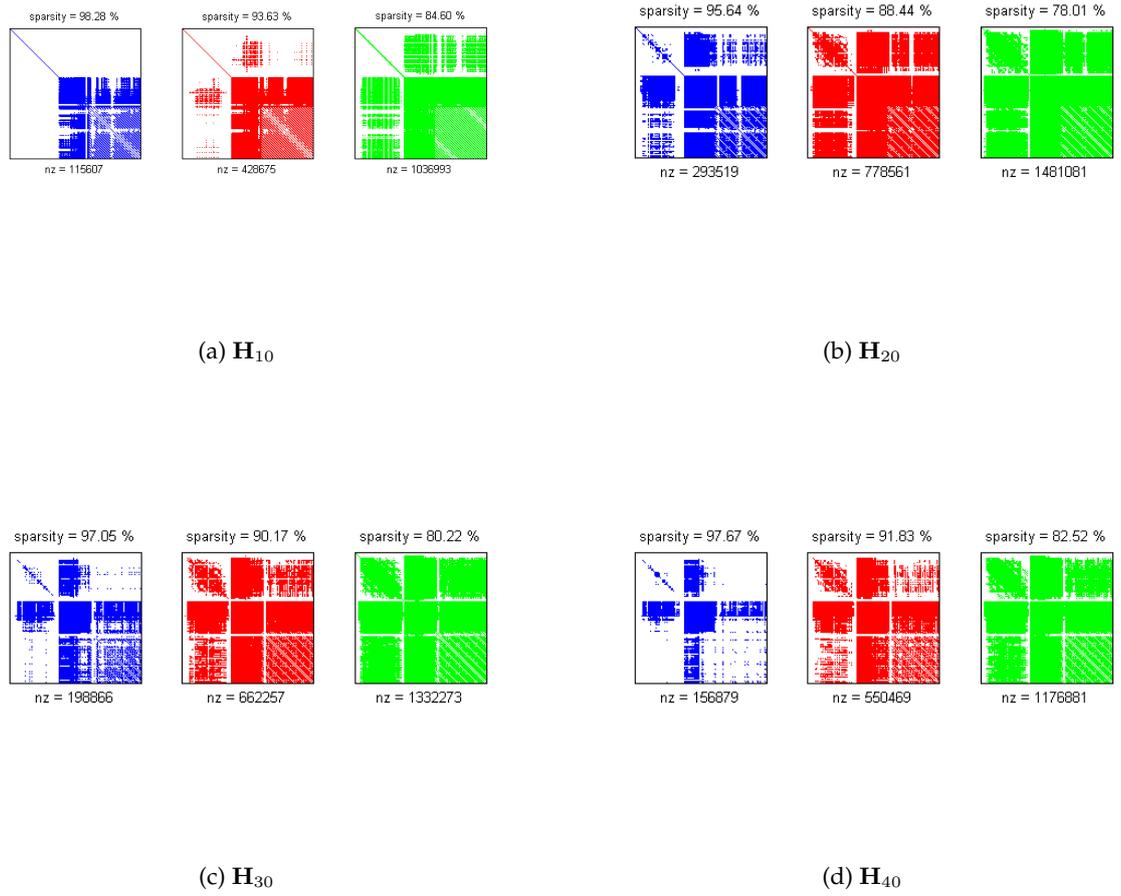
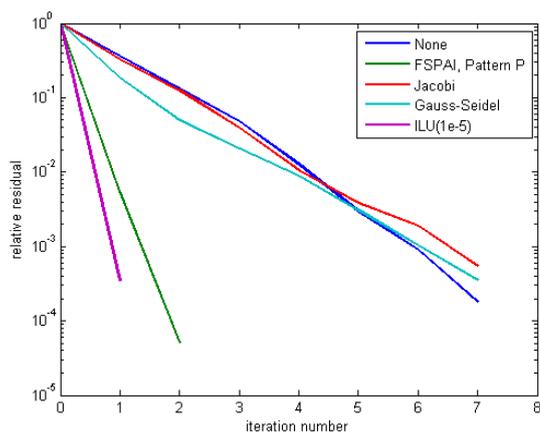
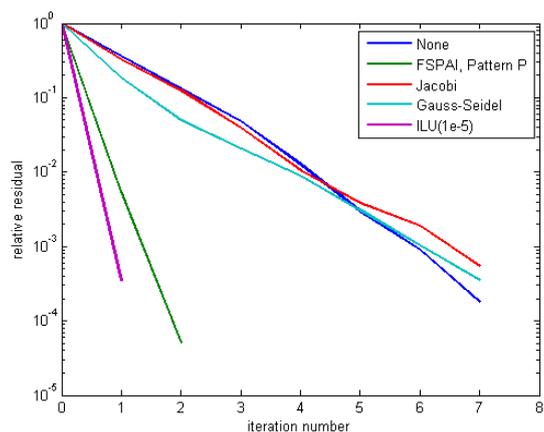


Figure B.2.: Chosen sparsity patterns P1,P2,P3 with decreasing sparsity for $\mathbf{H}_{10}, \mathbf{H}_{20}, \mathbf{H}_{30}, \mathbf{H}_{40}$

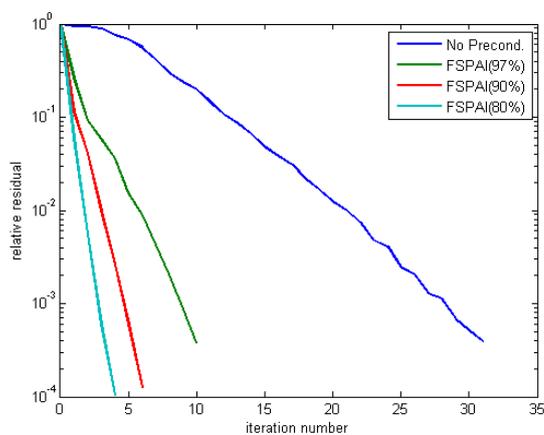
C. Results for different Hessians



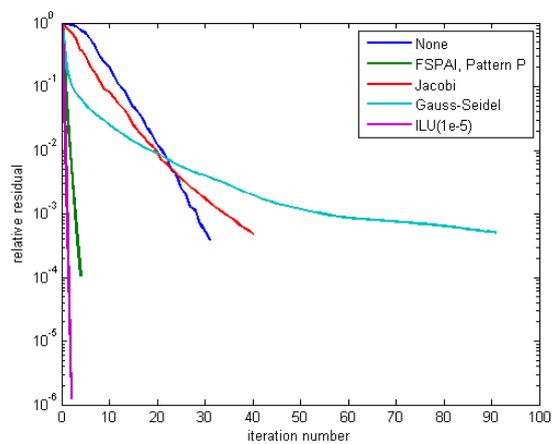
(a) Iteration 10, different pattern



(b) Iteration 10, different preconditioner

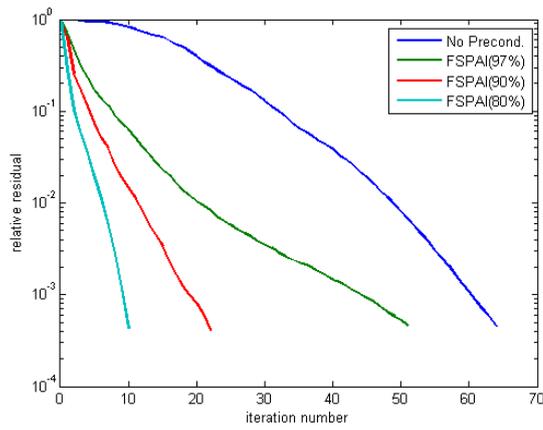


(c) Iteration 20, different pattern

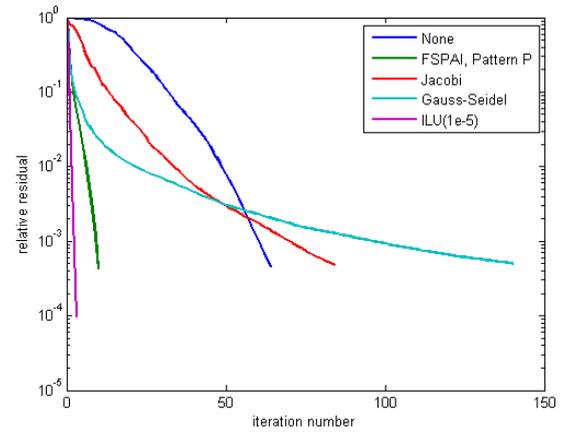


(d) Iteration 20, different preconditioner

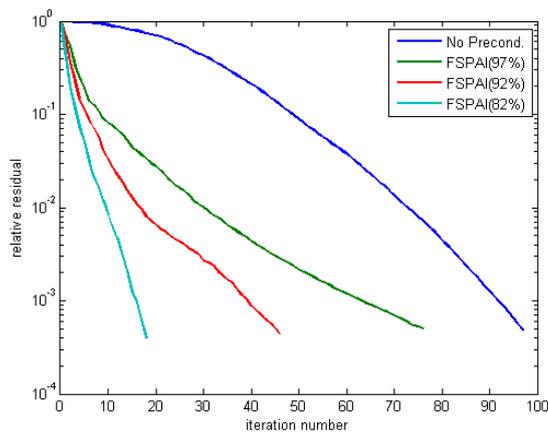
Figure C.1.: Results of the preconditioning for different preconditioner applied to \mathbf{H}_{10} and \mathbf{H}_{20}



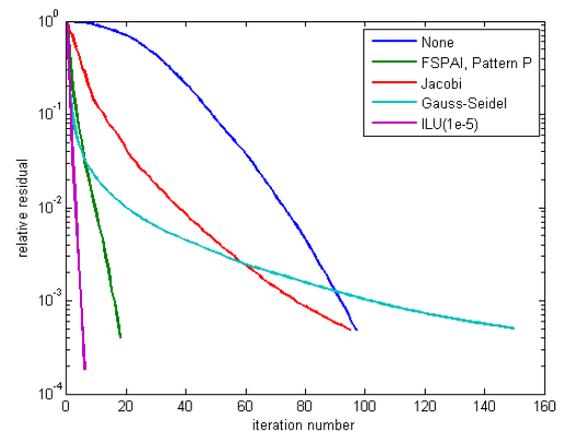
(a) Iteration 30, different pattern



(b) Iteration 30, different preconditioner



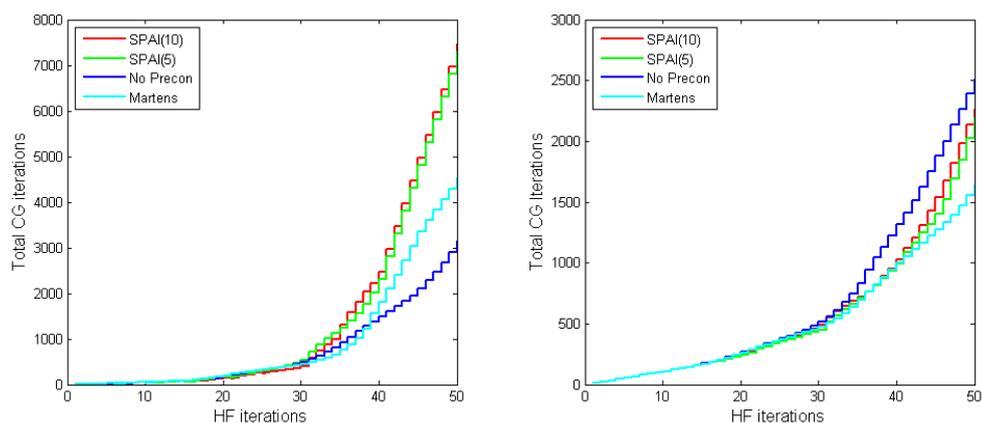
(c) Iteration 40, different pattern



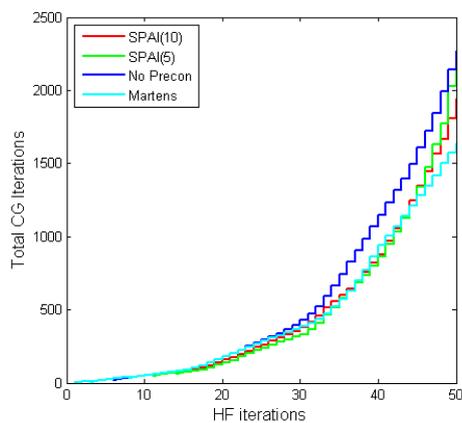
(d) Iteration 40, different preconditioner

Figure C.2.: Results of the preconditioning for different preconditioner applied to \mathbf{H}_{30} and \mathbf{H}_{40}

D. CG results



(a) HF optimization using a residual based stopping criterion (b) HF optimization using Martens quadratic based stopping criterion



(c) HF optimization using a combination of a residual based stopping criterion and Martens quadratic based stopping criterion

Figure D.1.: Results of the HF optimization using different stopping criteria for the inner iteration

Bibliography

- [1] R. BARRETT, M. BERRY, T. F. CHAN, J. DEMMEL, J. DONATO, J. DONGARRA, V. EIJKHOUT, R. POZO, C. ROMINE, and H. V. DER VORST, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*, SIAM, Philadelphia, PA, 1994.
- [2] Y. BENGIO and X. GLOROT, *Understanding the difficulty of training deep feedforward neural networks*, in Proceedings of AISTATS 2010, vol. 9, May 2010, pp. 249–256.
- [3] M. BENZI, *Preconditioning techniques for large linear systems: a survey*, J. Comput. Phys., 182, November 2002, pp. 418–477.
- [4] M. BENZI, J. K. CULLUM, and M. TUMA, *Robust approximate inverse preconditioning for the conjugate gradient method*, SIAM J. SCI. COMPUT, 22(4), 2000, pp. 1318–1332.
- [5] M. BENZI, L. GIRAUD, and G. ALLÉON, *Sparse Approximate Inverse Preconditioning For Dense Linear Systems Arising In Computational Electromagnetics*, Numerical Algorithms, 16, 1997, pp. 1–15.
- [6] M. BENZI, C. D. MEYER, and M. TUMA, *A Sparse Approximate Inverse Preconditioner For The Conjugate Gradient Method*, SIAM J. Sci. Comput, 17, 1996, pp. 1135–1149.
- [7] C. BISHOP, *Exact Calculation of the Hessian Matrix for the Multilayer Perceptron*, Neural Computation, 4(4), 1992, pp. 494–501.
- [8] C. M. BISHOP, *Pattern Recognition and Machine Learning (Information Science and Statistics)*, Springer, 1 ed., 2007.
- [9] O. CHAPELLE and D. ERHAN, *Improved Preconditioner for Hessian Free Optimization*, in NIPS Workshop on Deep Learning and Unsupervised Feature Learning, 2011.
- [10] K. CHEN, *Matrix Preconditioning Techniques and Applications (Cambridge Monographs on Applied and Computational Mathematics)*, Cambridge University Press, July 2005.
- [11] P. CHEN, *Hessian Matrix vs. Gauss-Newton Hessian Matrix*, SIAM J. Numer. Anal., 49, July 2011, pp. 1417–1435.

- [12] S. C. EISENSTAT, H. F. WALKER, S. C. EISENSTATT, HOMER, and F. WALKER, *Choosing the Forcing Terms in an Inexact Newton Method*, SIAM J. Sci. Comput, 17, 1994, pp. 16–32.
- [13] L. F. ESCUDERO, *On diagonally preconditioning the truncated Newton method for super-scale linearly constrained nonlinear programming*, European Journal of Operational Research, 17(3), September 1984, pp. 401–414.
- [14] M. FAIRBANK and E. ALONSO, *Efficient Calculation of the Gauss-Newton Approximation of the Hessian Matrix in Neural Networks*, Neural Computation, 24, 2012, pp. 607–610.
- [15] G. FASANO and M. ROMA, *An Approximate Inverse Preconditioner in Truncated Newton Methods for Large Scale Optimization*, Communications to SIMAI Congress, 1(0), 2007.
- [16] I. FODOR, *A Survey of Dimension Reduction Techniques*, tech. rep., 2002.
- [17] S. HAYKIN, *Neural networks and learning machines*, no. Bd. 10 in Neural networks and learning machines, Prentice Hall, 2009.
- [18] G. HINTON and R. SALAKHUTDINOV, *Reducing the Dimensionality of Data with Neural Networks*, Science, 313(5786), 2006, pp. 504 – 507.
- [19] T. HUCKLE, *Approximate Sparsity Patterns for the Inverse of a Matrix and Preconditioning*, Applied Numerical Mathematics, 30, 1999, pp. 291–303.
- [20] T. HUCKLE, *Factorized Sparse Approximate Inverses for Preconditioning*, J. Supercomput., 25(2), June 2003, pp. 109–117.
- [21] T. HUCKLE and A. KALLISCHKO, *Frobenius Norm Minimization and Probing for Preconditioning*, International Journal of Computer Mathematics, 84(8), Aug. 2007, pp. 1225–1248.
- [22] T. HUCKLE, A. KALLISCHKO, A. ROY, M. SEDLACEK, and T. WEINZIERL, *An efficient parallel implementation of the MSPAI preconditioner*, Parallel Computing, 36(5-6), 2010, pp. 273–284.
- [23] Q. LE, J. NGIAM, A. COATES, A. LAHIRI, B. PROCHNOW, and A. NG, *On optimization methods for deep learning*, in Proceedings of the 28th International Conference on Machine Learning (ICML-11), L. Getoor and T. Scheffer, eds., ICML '11, New York, NY, USA, June 2011, ACM, pp. 265–272.
- [24] C.-J. LIN and R. SAIGAL, *An Incomplete Cholesky Factorization for Dense Matrices*, Applied Numerical Mathematics, 536, p. 2000.
- [25] J. MARTENS, *Deep learning via Hessian-free optimization.*, in ICML, J. Fürnkranz and T. Joachims, eds., Omnipress, 2010, pp. 735–742.

- [26] J. MARTENS and I. SUTSKEVER, *Learning Recurrent Neural Networks with Hessian-Free Optimization*, in Proceedings of the 28th International Conference on Machine Learning (ICML-11), L. Getoor and T. Scheffer, eds., ICML '11, New York, NY, USA, June 2011, ACM, pp. 1033–1040.
- [27] S. G. NASH, *Truncated-newton methods*, PhD thesis, Stanford, CA, USA, 1982. AAI8220515.
- [28] S. G. NASH, *A survey of truncated-Newton methods*, J. Comput. Appl. Math., 124(1-2), Dec. 2000, pp. 45–59.
- [29] J. NOCEDAL and S. J. WRIGHT, *Numerical Optimization*, Springer, New York, 2nd ed., 2006.
- [30] B. A. PEARLMUTTER, *Fast Exact Multiplication by the Hessian*, Neural Computation, 6, 1994, pp. 147–160.
- [31] N. N. SCHRAUDOLPH, *Fast curvature matrix-vector products for second-order gradient descent*, Neural Comput., 14, July 2002, pp. 1723–1738.
- [32] C. SIEFERT, ERIC, and D. STURLER, *Probing Methods for saddle-point problems*, 2005.
- [33] A. VAN DER SLUIS and H. A. VAN DER VORST, *The rate of convergence of conjugate gradients*, Numer. Math., 48(5), May 1986, pp. 543–560.
- [34] O. VINYALS and D. POVEY, *Krylov Subspace Descent for Deep Learning*, 2011.