Deep Learning via Hessian-free Optimization

James Martens

University of Toronto

August 13, 2010

Computer Science
UNIVERSITY OF TORONTO
Gradient descent is bad at learning deep nets

The common experience:
- gradient descent gets much slower as the depth increases
Gradient descent is bad at learning deep nets

The common experience:

- gradient descent gets much slower as the depth increases
- large enough depth → learning to slow to a crawl or even “stops” → severe under-fitting (poor performance on the training set)
Gradient descent is bad at learning deep nets

The common experience:

- gradient descent gets much slower as the depth increases
- large enough depth $\rightarrow$ learning to slow to a crawl or even “stops” $\rightarrow$ severe under-fitting (poor performance on the training set)
- “vanishing-gradients problem”: error signal decays as it is backpropagated
Gradient descent is bad at learning deep nets

The common experience:

- gradient descent gets much slower as the depth increases
- large enough depth $\rightarrow$ learning to slow to a crawl or even "stops" $\rightarrow$ severe under-fitting (poor performance on the training set)
- "vanishing-gradients problem": error signal decays as it is backpropagated

- the gradient is tiny for weights in early layers

```
Input Layer  →  Intermediate Layer  →  Intermediate Layer  →  Output Layer
```

James Martens (U of T)
Gradient descent is bad at deep learning (cont.)

Two hypotheses for why gradient descent fails:

- increased frequency and severity of bad local minima:

\[ f(x, y) = (1 - x)^2 + 100(y - x^2)^2 \]
Gradient descent is bad at deep learning (cont.)

Two hypotheses for why gradient descent fails:

- increased frequency and severity of bad local minima:

- pathological curvature, like the type seen in the well-known Rosenbrock function:

  \[ f(x, y) = (1 - x)^2 + 100(y - x^2)^2 \]
Some early attempts address the vanishing gradients/pathological curvature issue:

**Momentum**
- average of the previous gradients with exponential decay
- physical analogy: builds “momentum” while descending down narrow valleys

**Adaptive learning rates (”R-prop”)**
- attempts to address the “vanishing gradients” problem directly
- individual parameters have learning rates that are adapted dynamically
- like a heuristically computed diagonal Hessian approximation
Pre-training for deep auto-encoders

(from Hinton and Salakhutdinov, 2006)
Pre-training (cont.)

- doesn’t generalize to all the sorts of deep-architectures we might wish to train
- still requires a classical optimization algorithm to “fine-tune” the parameters
- does it get full power out of deep auto-encoders?

(from Hinton and Salakhutdinov, 2006)
2nd-order optimization

If pathological curvature is the problem, this could be the solution.
2nd-order optimization

If pathological curvature is the problem, this could be the solution

**General framework**

- model the objective function by the local approximation:

\[
f(\theta + p) \approx q_\theta(p) \equiv f(\theta) + \nabla f(\theta)^\top p + \frac{1}{2} p^\top B p
\]

where \( B \) is a matrix which quantifies curvature
If pathological curvature is the problem, this could be the solution

**General framework**

- model the objective function by the local approximation:

\[
q_{\theta}(p) \equiv f(\theta) + \nabla f(\theta)^T p + \frac{1}{2} p^T B p
\]

where \( B \) is a matrix which quantifies curvature

- in Newton’s method, \( B = H \) or \( H + \lambda I \)
2nd-order optimization

If pathological curvature is the problem, this could be the solution

General framework

- model the objective function by the local approximation:

\[ f(\theta + p) \approx q_\theta(p) \equiv f(\theta) + \nabla f(\theta)^T p + \frac{1}{2} p^T B p \]

where \( B \) is a matrix which quantifies curvature

- in Newton’s method, \( B = H \) or \( H + \lambda I \)

- fully optimizing \( q_\theta(p) \) this w.r.t. \( p \) gives: \( p = -B^{-1} \nabla f(\theta) \)

- update is: \( \theta \leftarrow \theta + \alpha p \) for some \( \alpha \leq 1 \) determined by a line search
The importance of curvature (cont.)

Cartoon example of pathological curvature: the long narrow valley

- consider the following example where low and high-curvature directions co-occur. Using gradient descent gives one of the following 2 undesirable behaviors:

  large learning rate: high curvature directions pursued too far, undesirable “bouncing” behavior

  small learning rate: progress along low curvature directions is far too slow
Pathological curvature in deep-nets

- Suppose we have 2 *nearly* identical units (i.e. nearly identical weights and biases). Let $i$ and $j$ be the two red weights. Let $d$ direction with $d_k = \delta_{ik} - \delta_{jk}$. $d$ is a direction which differentiates these weights.
- Then the reduction is low: $-\nabla^T f d = (\nabla f)_j - (\nabla f)_i \approx 0$
- But so is the curvature: $d^T H d = (H_{ii} - H_{ij}) + (H_{jj} - H_{ji}) \approx 0 + 0 = 0$

**Left:** Neural net with nearly identical units (in the middle layer). Two weights with the same color have *nearly* identical values.

**Right:** Graphical representation of $d$
Vanishing Curvature

- define the direction $d$ by $d_k = \delta_{ik}$
- low reduction along $d$: $\mathbf{d}^\top \nabla f = - (\nabla f)_i \approx 0$
- but also low curvature: $\mathbf{d}^\top \mathbf{H} \mathbf{d} = - H_{ii} = \frac{\partial^2 f}{\partial \theta_i^2} \approx 0$

- so a 2nd-order optimizer will pursue $d$ at a reasonable rate, an elegant solution to the vanishing gradient problem of 1st-order optimizers
Practical Considerations for 2nd-order optimization

Hessian size problem

- for machine learning models the number of parameter \( N \) can be very large
- we can’t possibly calculate or even store a \( N \times N \) matrix, let alone invert one
Practical Considerations for 2nd-order optimization

Hessian size problem
- for machine learning models the number of parameters $N$ can be very large
- we can’t possibly calculate or even store a $N \times N$ matrix, let alone invert one

Quasi-Newton Methods
- non-linear conjugate gradient (NCG) - a hacked version of the quadratic optimizer linear CG
- limited-memory BFGS (L-BFGS) - a low rank Hessian approximation
- approximate diagonal or block-diagonal Hessian

Unfortunately these don’t seem to resolve the deep-learning problem
Hessian-free optimization

- a quasi-newton method that uses no low-rank approximations
- named 'free' because we never explicitly compute $B$
Hessian-free optimization

- a quasi-newton method that uses no low-rank approximations
- named 'free' because we never explicitly compute $B$

First motivating observation

- it is relatively easy to compute the matrix-vector product $Hv$ for an arbitrary vectors $v$
Hessian-free optimization

- a quasi-newton method that uses no low-rank approximations
- named 'free' because we never explicitly compute $B$

First motivating observation

- it is relatively easy to compute the matrix-vector product $Hv$ for an arbitrary vectors $v$
- e.g. use finite differences to approximate the limit:

$$Hv = \lim_{\epsilon \to 0} \frac{\nabla f(\theta + \epsilon v) - \nabla f(\theta)}{\epsilon}$$
Hessian-free optimization

- a quasi-newton method that uses no low-rank approximations
- named 'free' because we never explicitly compute $B$

First motivating observation

- it is relatively easy to compute the matrix-vector product $Hv$ for an arbitrary vector $v$
- e.g. use finite differences to approximate the limit:

$$Hv = \lim_{\epsilon \to 0} \frac{\nabla f(\theta + \epsilon v) - \nabla f(\theta)}{\epsilon}$$

- $Hv$ is computed for the exact value of $H$, there is no low-rank or diagonal approximation here!
Hessian-free optimization (cont.)

Second motivating observation

- linear conjugate gradient (CG) minimizes positive definite quadratic cost functions using only matrix-vector products.

These are related but different! We actually care about the quadratic, so this is good.
Second motivating observation

- linear conjugate gradient (CG) minimizes positive definite quadratic cost functions using only matrix-vector products
- more often seen in the context of solving large sparse systems
Second motivating observation

- linear conjugate gradient (CG) minimizes positive definite quadratic cost functions using only matrix-vector products
- more often seen in the context of solving large sparse systems
- directly minimizes the quadratic $q \equiv \frac{1}{2} p^\top B p + g^\top p$ and not the residual $\|Bp + g\|^2 \rightarrow$ these are related but different!
Hessian-free optimization (cont.)

Second motivating observation

- linear conjugate gradient (CG) minimizes positive definite quadratic cost functions using only matrix-vector products
- more often seen in the context of solving large sparse systems
- directly minimizes the quadratic \( q \equiv \frac{1}{2} p^\top B p + g^\top p \) and not the residual \( \|Bp + g\|^2 \rightarrow \) these are related but different!
- but we actually care about the quadratic, so this is good
Hessian-free optimization (cont.)

Second motivating observation

- linear conjugate gradient (CG) minimizes positive definite quadratic cost functions using only matrix-vector products
- more often seen in the context of solving large sparse systems
- directly minimizes the quadratic $q \equiv \frac{1}{2} p^\top B p + g^\top p$ and not the residual $\|Bp + g\|^2 \rightarrow$ these are related but different!
- but we actually care about the quadratic, so this is good
- requires $N = \text{dim}(\theta)$ iterations to converge in general, but makes a lot of progress in far fewer iterations than that
Standard Hessian-free Optimization

Pseudo-code for a simple variant of damped Hessian-free optimization:

1: for \( n = 1 \) to max-epochs do
2: compute gradient \( g_n = \nabla f(\theta_n) \)
3: choose/adapt \( \lambda_n \) according to some heuristic
4: define the function \( B_n(v) = Hv + \lambda_n v \)
5: \( p_n = \text{CGMinimize}(B_n, -g_n) \)
6: \( \theta_{n+1} = \theta_n + p_n \)
7: end for

In addition to choosing \( \lambda_n \), the stopping criterion for the CG algorithm is a critical detail.
Common variants of the HF approach

Basic/naive

- $\lambda_n = 0$, CG iterations stopped when residual $\|Bp + g\|$ reaches some error tolerance or when negative curvature is detected

CG-Steihaug

- $\lambda_n = 0$ and instead maintain a heuristically adjusted trust region
- when the iterates produced by the inner CG loop leave the trust region the loops terminates

Trust-region Newton-Lanczos Method

- $\lambda_n$ is (very expensively) computed to give match a given trust region radius
- robust even when the Hessian is indefinite
A new variant is required

- **the bad news**: common variants of HF (e.g. Steihaug) don't work particularly well for neural networks
A new variant is required

- **the bad news**: common variants of HF (e.g. Steihaug) don't work particular well for neural networks

- there are many aspects of the algorithm that are ill-defined in the basic approach which we need to address:
  - how can deal with negative curvature?
  - how should we choose $\lambda$?
  - how can we handle large data-sets
  - when should we stop the CG iterations?
  - can CG be accelerated?
Pearlmutter’s R-operator method

- finite-difference approximations are undesirable for many reasons
Pearlmutter’s R-operator method

- finite-difference approximations are undesirable for many reasons
- there is a better way to compute $Hv$ due to Pearlmutter (1994)
Pearlmutter’s R-operator method

- finite-difference approximations are undesirable for many reasons
- there is a better way to compute $Hv$ due to Pearlmutter (1994)
- similar cost to a gradient computation
Pearlmutter’s R-operator method

- finite-difference approximations are undesirable for many reasons
- there is a better way to compute $Hv$ due to Pearlmutter (1994)
- similar cost to a gradient computation
- for neural nets, no extra non-linear functions need to be evaluated
Pearlmutter’s R-operator method

- finite-difference approximations are undesirable for many reasons
- there is a better way to compute $Hv$ due to Pearlmutter (1994)
- similar cost to a gradient computation
- for neural nets, no extra non-linear functions need to be evaluated
- technique generalizes to almost any twice-differentiable function that is tractable to compute
Pearlmutter’s R-operator method

- finite-difference approximations are undesirable for many reasons
- there is a better way to compute $Hv$ due to Pearlmutter (1994)
- similar cost to a gradient computation
- for neural nets, no extra non-linear functions need to be evaluated
- technique generalizes to almost any twice-differentiable function that is tractable to compute
- can be automated (like automatic differentiation)
Forwards and backwards pass to compute the gradient

\[ \theta = (W_1, b_1, W_2, b_2, \ldots, W_L, b_L) \]

1: \( y_1 = \text{in} \)
2: \textbf{for} \ i = 1 \ \textbf{to} \ L \ \textbf{do} 
3: \quad x_i = W_i y_i + b_i 
4: \quad y_i = \sigma(x_i) 
5: \textbf{end for} 
6: \textbf{for} \ i = L \ \textbf{down to} \ 1 \ \textbf{do} 
7: \quad \textbf{if} \ i < L \ \textbf{then} 
8: \quad \quad \frac{dE}{dx_i} = \frac{dE}{dx_{i+1}} \odot y_{i+1} \odot (1 - y_{i+1}) 
9: \quad \textbf{else} 
10: \quad \quad \frac{dE}{dx_i} = \text{out} - y_{i+1} 
11: \quad \textbf{end if} 
12: \quad \frac{dE}{dy_i} = W_i^T \frac{dE}{dx_i} 
13: \quad \frac{dE}{dW_i} = \frac{dE}{dx_i} y_i^T 
14: \quad \frac{dE}{db_i} = \frac{dE}{dx_i} 
15: \textbf{end for}
The same code with the R-operator applied computes $H\nu$

$$\nu = (V_1, c_1, ..., V_L, c_L), \quad H\nu = (R\{\frac{dE}{dW_1}\}, R\{\frac{dE}{db_1}\}, ..., R\{\frac{dE}{dW_L}\}, R\{\frac{dE}{db_L}\})$$

1: $R\{y_1\} = 0$
2: for $i = 1$ to $L$ do
3: $R\{x_i\} = W_i R\{y_i\} + V_i y_i + c_i$
4: $R\{y_i\} = R\{x_i\} \circ y_{i+1} \circ (1 - y_{i+1})$
5: end for
6: for $i = L$ down to 1 do
7: if $i < L$ then
8: $R\{\frac{dE}{dx_i}\} = R\{\frac{dE}{dx_{i+1}}\} \circ y_{i+1} \circ (1 - y_{i+1}) + \frac{dE}{dx_{i+1}} \circ R\{y_{i+1}\} \circ (1 - 2y_{i+1})$
9: else
10: $R\{\frac{dE}{dx_i}\} = -R\{y_{i+1}\}$
11: end if
12: $R\{\frac{dE}{dy_i}\} = V_i^T \frac{dE}{dx_i} + W_i^T R\{\frac{dE}{dx_i}\}$
13: $R\{\frac{dE}{dW_i}\} = R\{\frac{dE}{dx_i}\} y_i^T + \frac{dE}{dx_i} R\{y_i\}^T$
14: $R\{\frac{dE}{db_i}\} = R\{\frac{dE}{dx_i}\}$
15: end for
The Gauss-Newton Matrix (G)

- a well-known alternative to the Hessian that is guaranteed to be positive semi-definite - thus no negative curvature!

\[ G = J^T J \] (\(J\) is the Jacobian of the output units w.r.t. \(\theta\))

Can be generalized beyond just least squares to neural nets with "matching" loss functions and output non-linearities (Schraudolph 2002), e.g. logistic units with cross-entropy error works much better in practice than Hessian or other curvature matrices (e.g. empirical Fisher). We can compute \(G\) using an algorithm similar to the one for \(H\).
The Gauss-Newton Matrix \((G)\)

- a well-known alternative to the Hessian that is guaranteed to be positive semi-definite - thus no negative curvature!

- usually applied to non-linear least-squares problems where it is given by \(G = J^T J\) (\(J\) is the Jacobian of the output units w.r.t. \(\theta\))

- can be generalized beyond just least squares to neural nets with “matching” loss functions and output non-linearities (Schraudolph 2002)
  - e.g. logistic units with cross-entropy error
The Gauss-Newton Matrix ($G$)

- a well-known alternative to the Hessian that is guaranteed to be positive semi-definite - thus no negative curvature!

- usually applied to non-linear least-squares problems where it is given by $G = J^T J$ ($J$ is the Jacobian of the output units w.r.t. $\theta$)

- can be generalized beyond just least squares to neural nets with “matching” loss functions and output non-linearities (Schraudolph 2002)
  - e.g. logistic units with cross-entropy error

- works much better in practice than Hessian or other curvature matrices (e.g. empirical Fisher)
The Gauss-Newton Matrix (G)

- a well-known alternative to the Hessian that is guaranteed to be positive semi-definite - thus no negative curvature!

- usually applied to non-linear least-squares problems where it is given by $G = J^T J$ ($J$ is the Jacobian of the output units w.r.t. $\theta$)

- can be generalized beyond just least squares to neural nets with “matching” loss functions and output non-linearities (Schraudolph 2002)
  - e.g. logistic units with cross-entropy error

- works much better in practice than Hessian or other curvature matrices (e.g. empirical Fisher)

- and we can compute $Gv$ using an algorithm similar to the one for $ Hv$
CG stopping conditions

- CG is only guaranteed to converge after $N$ (size of parameter space) iterations $\rightarrow$ we can't always run it to convergence
CG stopping conditions

- CG is only guaranteed to converge after $N$ (size of parameter space) iterations $\rightarrow$ we can’t always run it to convergence

- the standard stopping criterion used in most versions of HF is $\|r\| < \min\left(\frac{1}{2}, \|g\|^{\frac{1}{2}}\right)\|g\|$ where $r = Bp + g$ is the “residual”
CG stopping conditions

- CG is only guaranteed to converge after $N$ (size of parameter space) iterations → we can’t always run it to convergence

- the standard stopping criterion used in most versions of HF is
  $$\|r\| < \min\left(\frac{1}{2}, \|g\|^\frac{1}{2}\right)\|g\|$$
  where $r = Bp + g$ is the “residual”

- strictly speaking $\|r\|$ is not the quantity that CG minimizes, nor is it the one we really care about

---

James Martens (U of T)

Deep Learning via HF

August 13, 2010 21 / 29
we found that terminating CG once the relative per-iteration reduction rate fell below some tolerance $\epsilon$ worked best

$$\frac{\Delta q}{q} < \epsilon$$

($\Delta q$ is the change in the quadratic model averaged over some window of the last $k$ iterations of CG)
Handling large datasets

- each iteration of CG requires the evaluation of the product $Bv$ for some $v$
Handling large datasets

- each iteration of CG requires the evaluation of the product $Bv$ for some $v$
- naively this requires a pass over the training data-set
Handling large datasets

- each iteration of CG requires the evaluation of the product $Bv$ for some $v$
- naively this requires a pass over the training data-set
- but for a sufficiently large subset of the training data - sufficient to capture enough useful curvature information
Handling large datasets

- each iteration of CG requires the evaluation of the product $Bv$ for some $v$

- naively this requires a pass over the training data-set

- but for a sufficiently large subset of the training data - sufficient to capture enough useful curvature information

- size is related to model and qualitative aspects of the dataset, but critically not its size
  - for very large datasets, mini-batches might be a tiny fraction of the whole
Handling large datasets

- each iteration of CG requires the evaluation of the product $Bv$ for some $v$
- naively this requires a pass over the training data-set
- but for a sufficiently large subset of the training data - sufficient to capture enough useful curvature information
- size is related to model and qualitative aspects of the dataset, but critically not its size
  - for very large datasets, mini-batches might be a tiny fraction of the whole
- gradient and line-searches can be computed using even larger mini-batches since they are needed much less often
Damping the curvature matrix

- we don’t completely trust the quadratic model as an approximation

\[ B = G + \lambda I \]

where \( \lambda \) is adjusted at each (outer) iteration using the standard Levenburg-Marquardt style heuristic:

\[
\rho \leftarrow f(\theta + p) - f(\theta) - \frac{q}{\theta(0)}
\]

if \( \rho < \frac{1}{4} \)

else if \( \rho > \frac{3}{4} \)

\[ \lambda \leftarrow \frac{3}{2} \lambda \]

end if
Damping the curvature matrix

- we don’t completely trust the quadratic model as an approximation
- a good way to account for this is to “damp” $B$

$$B = G + \lambda I$$

where $\lambda$ is adjusted at each (outer) iteration using the standard Levenburg-Marquardt style heuristic:

$$\rho \leftarrow f(\theta + p) - f(\theta)$$

$$q_{\theta}(p) - q_{\theta}(0)$$

if $\rho < 1/4$ then

$$\lambda \leftarrow 3/2 \lambda$$

else if $\rho > 3/4$ then

$$\lambda \leftarrow 2/3 \lambda$$

end if
Damping the curvature matrix

- we don’t completely trust the quadratic model as an approximation
- a good way to account for this is to “damp” $B$
- we take $B = G + \lambda I$ where $\lambda$ is adjusted at each (outer) iteration using the standard Levenburg-Marquardt style heuristic:

$$\rho \leftarrow \frac{f(\theta + p) - f(\theta)}{q_\theta(p) - q_\theta(0)}$$
Damping the curvature matrix

- we don’t completely trust the quadratic model as an approximation
- a good way to account for this is to “damp” $B$
- we take $B = G + \lambda I$ where $\lambda$ is adjusted at each (outer) iteration using the standard Levenburg-Marquardt style heuristic:

$$
\rho \leftarrow \frac{f(\theta+p)-f(\theta)}{q_\theta(p)-q_\theta(0)}
$$

- if $\rho < \frac{1}{4}$ then
  - $\lambda \leftarrow \frac{3}{2} \lambda$
- else if $\rho > \frac{3}{4}$ then
  - $\lambda \leftarrow \frac{2}{3} \lambda$
- end if
the normal damping term can be interpreted as putting an $\ell_2$ prior on the parameters that says “don’t change”:

$$f(\theta + p) \approx q_\theta(p) \equiv f(\theta) + \nabla f(\theta)^\top p + \frac{1}{2} p^\top (G + \lambda I)p$$

$$= f(\theta) + \nabla f(\theta)^\top p + \frac{1}{2} p^\top Gp + \frac{\lambda}{2} \|p\|^2$$
Structural damping

- the normal damping term can be interpreted as putting an $\ell_2$ prior on the parameters that says “don’t change”:

$$f(\theta + p) \approx q_\theta(p) \equiv f(\theta) + \nabla f(\theta)^\top p + \frac{1}{2}p^\top (G + \lambda I)p$$

$$= f(\theta) + \nabla f(\theta)^\top p + \frac{1}{2}p^\top Gp + \frac{\lambda}{2}||p||^2$$

- this treats all directions in parameter space “equally”
Structural damping

- the normal damping term can be interpreted as putting an $\ell_2$ prior on the parameters that says “don’t change”:

$$f(\theta + p) \approx q_\theta(p) \equiv f(\theta) + \nabla f(\theta)^\top p + \frac{1}{2} p^\top (G + \lambda I)p$$

$$= f(\theta) + \nabla f(\theta)^\top p + \frac{1}{2} p^\top Gp + \frac{\lambda}{2} \|p\|^2$$

- this treats all directions in parameter space “equally”

- however, some directions lead to large fluctuations in the hidden-unit activations whilst others have a much smaller effect
Structural damping

- the normal damping term can be interpreted as putting an $\ell_2$ prior on the parameters that says “don’t change”:

$$f(\theta + p) \approx q_\theta(p) \equiv f(\theta) + \nabla f(\theta)^\top p + \frac{1}{2} p^\top (G + \lambda I)p$$

$$= f(\theta) + \nabla f(\theta)^\top p + \frac{1}{2} p^\top Gp + \frac{\lambda}{2} \|p\|^2$$

- this treats all directions in parameter space “equally”

- however, some directions lead to large fluctuations in the hidden-unit activations whilst others have a much smaller effect

- for extremely non-linear models like Recurrent Neural Nets (RNNs) we expect this effect to be pronounced and so we would prefer to “damp” directions in a more intelligent way
so let’s put a “do not change” prior on the hidden unit activities $h_t$!
so let’s put a “do not change” prior on the hidden unit activities $h_t$!

for example, we could add the term:

$$\frac{\gamma}{2} \| h(\theta + p) - h(\theta) \|^2$$
so let's put a “do not change” prior on the hidden unit activities $h_t$!

for example, we could add the term:

$$
\frac{\gamma}{2} \| h(\theta + p) - h(\theta) \|^2
$$

unlike $\frac{\lambda}{2} \| p \|^2$ this term is not quadratic in $p$
so let’s put a “do not change” prior on the hidden unit activities $h_t$!

for example, we could add the term:

$$\frac{\gamma}{2} \| h(\theta + p) - h(\theta) \|^2$$

unlike $\frac{\lambda}{2} \| p \|^2$ this term is not quadratic in $p$
however, we can make it so by applying the usual Gauss-Newton approximation
- however, we can make it so by applying the usual Gauss-Newton approximation

- this gives the following contribution to $q$:

$$
\frac{\gamma}{2} p J_h^T J_h p
$$

where $J_h$ is the Jacobian of the hidden units w.r.t. the parameters
- however, we can make it so by applying the usual Gauss-Newton approximation

- this gives the following contribution to $q$:

  $$
  \frac{\gamma}{2} p^T J_h^T J_h p
  $$

  where $J_h$ is the Jacobian of the hidden units w.r.t. the parameters

- fortunately $J_h^T v$ occurs as an intermediate quantity in the algorithm for computing $Jv$
• however, we can make it so by applying the usual Gauss-Newton approximation

• this gives the following contribution to $q$:

$$\frac{\gamma}{2} p J_h^T J_h p$$

where $J_h$ is the Jacobian of the hidden units w.r.t. the parameters

• fortunately $J_h \nu$ occurs as an intermediate quantity in the algorithm for computing $J \nu$

• so it is a trivial matter to modify the algorithm include the term $\frac{\gamma}{2} p J_h^T J_h p$
Other enhancements

- using M-preconditioned CG with the diagonal preconditioner:

\[ M = \left[ \text{diag} \left( \sum_i \nabla f_i \odot \nabla f_i \right) + \lambda I \right]^\alpha \]

- initializing each run of the inner CG-loop from the solution found by the previous run
- carefully bounding and "back-tracking" the maximum number of CG steps to compensate for the effect of using mini-batches to compute the \( Bv \) products (see the paper for further details)
Other enhancements

- using M-preconditioned CG with the diagonal preconditioner:

\[ M = \left[ \text{diag} \left( \sum_i \nabla f_i \odot \nabla f_i \right) + \lambda I \right]^{\alpha} \]

- initializing each run of the inner CG-loop from the solution found by the previous run
Other enhancements

- using M-preconditioned CG with the diagonal preconditioner:

\[ M = \left[ \text{diag} \left( \sum_i \nabla f_i \odot \nabla f_i \right) + \lambda I \right]^\alpha \]

- initializing each run of the inner CG-loop from the solution found by the previous run

- carefully bounding and “back-tracking” the maximum number of CG steps to compensate for the effect of using mini-batches to compute the $Bv$ products
Other enhancements

- using M-preconditioned CG with the diagonal preconditioner:

\[ M = \left[ \text{diag} \left( \sum_i \nabla f_i \odot \nabla f_i \right) + \lambda I \right]^{\alpha} \]

- initializing each run of the inner CG-loop from the solution found by the previous run

- carefully bounding and “back-tracking” the maximum number of CG steps to compensate for the effect of using mini-batches to compute the \( Bv \) products

- (see the paper for further details)
Thank you for your attention