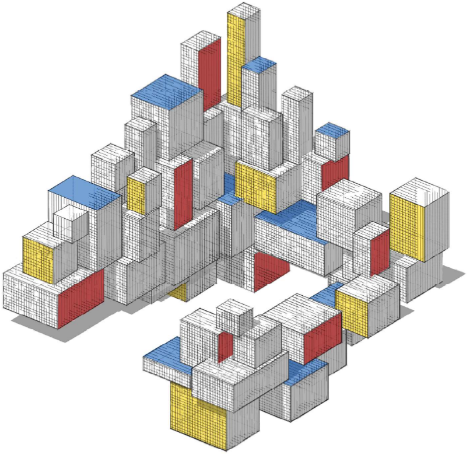
Training Neural Networks



Purpose

We discuss in more detail what is needed to train a feed-forward neural networks, including:

- Steepest descent
- Stochastic gradient descent
- Limited-Memory BFGS method

In what follows, the vectors $\boldsymbol{\delta}_t$ and $\boldsymbol{g}_t$ denote differences of vectors and gradients used in quasi-Newton methods, and should not be confused with the derivative $\boldsymbol{\delta}$ and the prediction function $\boldsymbol{g}$, discussed in earlier lectures.

Steepest Descent

Training a neural network requires the minimization of a training loss, $\ell_\tau(\boldsymbol{g}(\cdot \,|\, \boldsymbol{\theta})) = \frac{1}{n}\sum_{i=1}^{n} C_i(\boldsymbol{\theta})$.

If we can compute the gradient of $\ell_\tau(\boldsymbol{g}(\cdot \,|\, \boldsymbol{\theta}))$ via BP, then we can apply the steepest descent algorithm:

Starting from a guess $\boldsymbol{\theta}_1$, we iterate until convergence:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha_t\, \boldsymbol{u}_t, \qquad t = 1, 2, \ldots, \tag{1}$$

where $\boldsymbol{u}_t := \frac{\partial \ell_\tau}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_t)$ and α_t is the learning rate.

Here, $\boldsymbol{\theta} := \{\mathbf{W}_l, \boldsymbol{b}_l\}_{l=1}^{L}$ is a column vector of length $\sum_{l=1}^{L}(p_{l-1}p_l + p_l)$ that stores all the weight and bias parameters.

The advantage of organizing the computations in this way is that we can easily compute the learning rate α_t.

Algorithm 1: Training via Steepest Descent

Input: Training set $\tau = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^n$, initial weight matrices and bias vectors $\{\mathbf{W}_l, \boldsymbol{b}_l\}_{l=1}^L =: \boldsymbol{\theta}_1$, activation functions $\{\boldsymbol{S}_l\}_{l=1}^L$.

Output: The parameters of the trained learner.

```
1  $t \leftarrow 1,\ \boldsymbol{\delta} \leftarrow 0.1 \times \mathbf{1},\ \boldsymbol{u}_{t-1} \leftarrow \mathbf{0},\ \alpha \leftarrow 0.1$      // initialization
2  while stopping condition is not met do
3      Compute the gradient $\boldsymbol{u}_t = \frac{\partial \ell_\tau}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_t)$ using the BP algorithm.
4      $\boldsymbol{g} \leftarrow \boldsymbol{u}_t - \boldsymbol{u}_{t-1}$
5      if $\boldsymbol{\delta}^\top \boldsymbol{g} > 0$ then// check if Hessian is positive-definite
6          $\alpha \leftarrow \boldsymbol{\delta}^\top \boldsymbol{g} / \|\boldsymbol{g}\|^2$                    // Barzilai-Borwein
7      else
8          $\alpha \leftarrow 2 \times \alpha$      // failing positivity, do something
            heuristic
9      $\boldsymbol{\delta} \leftarrow -\alpha\, \boldsymbol{u}_t$
10     $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \boldsymbol{\delta}$
11     $t \leftarrow t + 1$
12 return $\boldsymbol{\theta}_t$ as the minimizer of the training loss
```

Stochastic Gradient Descent

Computing the gradient of the training loss via the BP algorithm requires averaging over all training examples. When the trainin set τ_n is very large, computation of the gradient $\partial \ell_{\tau_n} / \partial \boldsymbol{\theta}$ may be too costly.

In such cases, we may employ the stochastic gradient descent algorithm, where we view the training loss as an expectation:

$$\ell_\tau(\boldsymbol{g}(\cdot \,|\, \boldsymbol{\theta})) = \frac{1}{n} \sum_{k=1}^{n} \mathrm{Loss}(\boldsymbol{y}_k, \boldsymbol{g}(\boldsymbol{x}_k \,|\, \boldsymbol{\theta})) = \mathbb{E}\, \mathrm{Loss}(\boldsymbol{y}_K, \boldsymbol{g}(\boldsymbol{x}_K \,|\, \boldsymbol{\theta})),$$

where $\mathbb{P}[K = k] = 1/n$ for $k = 1, \ldots, n$.

We can thus approximate $\ell_\tau(\boldsymbol{g}(\cdot \,|\, \boldsymbol{\theta}))$ via a Monte Carlo estimator:

$$\widehat{\ell_\tau}(\boldsymbol{g}(\cdot \,|\, \boldsymbol{\theta})) := \frac{1}{N} \sum_{i=1}^{N} \mathrm{Loss}(\boldsymbol{y}_{K_i}, \boldsymbol{g}(\boldsymbol{x}_{K_i} \,|\, \boldsymbol{\theta})),$$

where $N \ll n$. The iid sample $K_1, \ldots, K_N$ is called a minibatch.

Second-order Methods

The steepest descent method

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha_t \boldsymbol{u}_t, \quad t = 1, 2, \ldots$$

only uses the information of the gradient vector $\boldsymbol{u}_t := \frac{\partial \ell_\tau}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_t)$ at the current candidate solution $\boldsymbol{\theta}_t$.

In trying to design a more efficient second-order optimization method, we may be tempted to use Newton's method:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{H}_t^{-1} \boldsymbol{u}_t, \quad t = 1, 2, \ldots$$

where $\mathbf{H}_t$ is the $d \times d$ matrix of second-order partial derivatives of $\ell_\tau(\boldsymbol{g}(\cdot \,|\, \boldsymbol{\theta}))$ at $\boldsymbol{\theta}_t$.

Second-order Methods

There are two problems with this approach:

1. While the computation of $\boldsymbol{u}_t$ via backpropagation typically costs $\mathcal{O}(d)$, the computation of $\mathbf{H}_t$ costs $\mathcal{O}(d^2)$.
2. Even if we have somehow computed $\mathbf{H}_t$ very fast, computing the search direction $\mathbf{H}_t^{-1}\,\boldsymbol{u}_t$ still incurs an $\mathcal{O}(d^3)$ cost.

Both of these considerations make Newton's method impractical for large d.

A practical alternative is to use a quasi-Newton method, in which we directly aim to approximate $\mathbf{H}_t^{-1}$ via a matrix $\mathbf{C}_t$ that satisfies the secant condition:

$$\mathbf{C}_t\, \boldsymbol{g}_{t-1} = \boldsymbol{\delta}_{t-1},$$

where $\boldsymbol{\delta}_t := \boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t$ and $\boldsymbol{g}_t := \boldsymbol{u}_{t+1} - \boldsymbol{u}_t$.

Broyden-Fletcher-Goldfarb-Shanno (BFGS) Method

An ingenious formula that generates a suitable sequence of approximating matrices $\{\mathbf{C}_t\}$ (each satisfying the secant condition) is the BFGS updating formula, which can be written as the recursion:

$$\mathbf{C}_{t+1} = \left(\mathbf{I} - \upsilon_t \, \boldsymbol{g}_t \boldsymbol{\delta}_t^\top\right)^\top \mathbf{C}_t \left(\mathbf{I} - \upsilon_t \, \boldsymbol{g}_t \boldsymbol{\delta}_t^\top\right) + \upsilon_t \, \boldsymbol{\delta}_t \boldsymbol{\delta}_t^\top, \quad \upsilon_t := (\boldsymbol{g}_t^\top \boldsymbol{\delta}_t)^{-1}. \tag{2}$$

This formula allows us to update $\mathbf{C}_{t-1}$ to $\mathbf{C}_t$ and then compute $\mathbf{C}_t \, \boldsymbol{u}_t$ in $\mathcal{O}(d^2)$ time.

While this quasi-Newton approach is better than the $\mathcal{O}(d^3)$ cost of Newton's method, it may be still too costly in large-scale applications.

Limited Memory BFGS

Instead, an approximate or limited memory BFGS updating can be achieved in $\mathcal{O}(d)$ time.

The idea is to store a few of the most recent pairs $\{\boldsymbol{\delta}_t, \boldsymbol{g}_t\}$ in order to evaluate its action on a vector $\boldsymbol{u}_t$ without explicitly constructing and storing $\mathbf{C}_t$ in computer memory.

This is possible, because updating $\mathbf{C}_0$ to $\mathbf{C}_1$ in (2) requires only the pair $\boldsymbol{\delta}_1, \boldsymbol{g}_1$, and similarly computing $\mathbf{C}_t$ from $\mathbf{C}_0$ only requires the history of the updates $\boldsymbol{\delta}_1, \boldsymbol{g}_1 \ldots, \boldsymbol{\delta}_t, \boldsymbol{g}_t$, which can be shown as follows.

BFGS Updating

Define the matrices $\mathbf{A}_t, \dots, \mathbf{A}_0$ via the backward recursion:

$$\mathbf{A}_t := \mathbf{I}, \quad \mathbf{A}_{j-1} := \left(\mathbf{I} - \upsilon_j\, \boldsymbol{g}_j \boldsymbol{\delta}_j^\top\right) \mathbf{A}_j, \quad j = 1, \dots, t,$$

and observe that $\boldsymbol{q}_j := \mathbf{A}_j\, \boldsymbol{u}$, $j = 0, \dots, t$ can be computed efficiently via the backward recursion starting with $\boldsymbol{q}_t = \boldsymbol{u}$:

$$\tau_j := \boldsymbol{\delta}_j^\top \boldsymbol{q}_j, \quad \boldsymbol{q}_{j-1} = \boldsymbol{q}_j - \upsilon_j \tau_j\, \boldsymbol{g}_j, \quad j = t, t-1, \dots, 1. \qquad (3)$$

In addition, define $\{\boldsymbol{r}_j\}$ via the recursion:

$$\boldsymbol{r}_0 := \mathbf{C}_0\, \boldsymbol{q}_0, \quad \boldsymbol{r}_j = \boldsymbol{r}_{j-1} + \upsilon_i \left(\tau_i - \boldsymbol{g}_j^\top \boldsymbol{r}_{j-1}\right) \boldsymbol{\delta}_j, \quad j = 1, \dots, t. \qquad (4)$$

At each iteration t, the BFGS updating formula (2) can be rewritten in the form:

$$\mathbf{C}_t = \mathbf{A}_{t-1}^\top \mathbf{C}_{t-1} \mathbf{A}_{t-1} + \upsilon_t\, \boldsymbol{\delta}_t \boldsymbol{\delta}_t^\top.$$

BFGS Updating

By iterating this recursion backwards to $\mathbf{C}_0$, we can write:

$$\mathbf{C}_t = \mathbf{A}_0^\top \mathbf{C}_0 \mathbf{A}_0 + \sum_{j=1}^{t} \upsilon_j \, \mathbf{A}_j^\top \boldsymbol{\delta}_j \boldsymbol{\delta}_j^\top \mathbf{A}_j,$$

that is, we can express $\mathbf{C}_t$ in terms of the initial $\mathbf{C}_0$ and the entire history of all BFGS values $\{\boldsymbol{\delta}_j, \boldsymbol{g}_j\}$.

Further, with the $\{\boldsymbol{q}_j, \boldsymbol{r}_j\}$ computed via (3) and (4), we can write:

$$\begin{aligned}
\mathbf{C}_t \, \boldsymbol{u} &= \mathbf{A}_0^\top \mathbf{C}_0 \, \boldsymbol{q}_0 + \sum_{j=1}^{t} \upsilon_j \left(\boldsymbol{\delta}_j^\top \boldsymbol{q}_j\right) \mathbf{A}_j^\top \boldsymbol{\delta}_j \\
&= \mathbf{A}_0^\top \boldsymbol{r}_0 + \upsilon_1 \tau_1 \mathbf{A}_1^\top \boldsymbol{\delta}_1 + \sum_{j=2}^{t} \upsilon_j \tau_j \mathbf{A}_j^\top \boldsymbol{\delta}_j \\
&= \mathbf{A}_1^\top \left[\left(\mathbf{I} - \upsilon_1 \boldsymbol{\delta}_1 \boldsymbol{g}_1^\top\right) \boldsymbol{r}_0 + \upsilon_1 \tau_1 \boldsymbol{\delta}_1\right] + \sum_{j=2}^{t} \upsilon_j \tau_j \mathbf{A}_j^\top \boldsymbol{\delta}_j .
\end{aligned}$$

BFGS Updating

Hence, from the definition of the $\{\boldsymbol{r}_j\}$ in (4), we obtain

$$\begin{aligned}\mathbf{C}_t \boldsymbol{u} &= \mathbf{A}_1^\top \boldsymbol{r}_1 + \sum_{j=2}^{t} \upsilon_j \tau_j \mathbf{A}_j^\top \boldsymbol{\delta}_j \\ &= \mathbf{A}_2^\top \boldsymbol{r}_2 + \sum_{j=3}^{t} \upsilon_j \tau_j \mathbf{A}_j^\top \boldsymbol{\delta}_j \\ &= \cdots = \mathbf{A}_t^\top \boldsymbol{r}_t + 0 = \boldsymbol{r}_t .\end{aligned}$$

Given $\mathbf{C}_0$ and the history of all recent BFGS values $\{\boldsymbol{\delta}_j, \boldsymbol{g}_j\}_{j=1}^{h}$, the computation of the quasi-Newton search direction $\boldsymbol{d} = -\mathbf{C}_h \boldsymbol{u}$ can be accomplished via the recursions (3) and (4) as summarized in Algorithm 2.

Algorithm 2: Limited-Memory BFGS Update

Input: BFGS history list $\{\boldsymbol{\delta}_j, \boldsymbol{g}_j\}_{j=1}^h$, initial $\mathbf{C}_0$, and input $\boldsymbol{u}$.

Output: $\boldsymbol{d} = -\mathbf{C}_h\,\boldsymbol{u}$, where

$$\mathbf{C}_h = \left(\mathbf{I} - \upsilon_j\,\boldsymbol{\delta}_j\boldsymbol{g}_j^\top\right)\mathbf{C}_{t-1}\left(\mathbf{I} - \upsilon_j\,\boldsymbol{g}_j\boldsymbol{\delta}_j^\top\right) + \upsilon_t\,\boldsymbol{\delta}_t\boldsymbol{\delta}_t^\top.$$

```
1 $\boldsymbol{q} \leftarrow \boldsymbol{u}$
2 for $i = h, h-1, \dots, 1$ do          // backward recursion to compute $\mathbf{A}_0\,\boldsymbol{u}$
3     $\upsilon_i \leftarrow \left(\boldsymbol{\delta}_i^\top \boldsymbol{g}_i\right)^{-1}$
4     $\tau_i \leftarrow \boldsymbol{\delta}_i^\top \boldsymbol{q}$
5     $\boldsymbol{q} \leftarrow \boldsymbol{q} - \upsilon_i\tau_i\,\boldsymbol{g}_i$
6 $\boldsymbol{q} \leftarrow \mathbf{C}_0\,\boldsymbol{q}$                          // compute $\mathbf{C}_0(\mathbf{A}_0\,\boldsymbol{u})$
7 for $i = 1, \dots, h$ do                 // compute recursion (4)
8     $\boldsymbol{q} \leftarrow \boldsymbol{q} + \upsilon_i(\tau_i - \boldsymbol{g}_i^\top\boldsymbol{q})\,\boldsymbol{\delta}_i$
9 return $\boldsymbol{d} \leftarrow -\boldsymbol{q}$, the value of $-\mathbf{C}_h\,\boldsymbol{u}$
```

Limited Memory BFGS

Note that if $\mathbf{C}_0$ is a diagonal matrix, say the identity matrix, then $\mathbf{C}_0\, \boldsymbol{q}$ is cheap to compute and the cost of running Algorithm 2 is $\mathcal{O}(h\, d)$.

Thus, for a fixed length of the BFGS history, the cost of the limited-memory BFGS updating grows linearly in d, making it a viable optimization algorithm in large-scale applications.

In summary, a quasi-Newton algorithm with limited-memory BFGS updating is given next.

Algorithm 3: Quasi-Newton with Limited-Memory BFGS

```
Input: $\tau = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^n$, $\{\mathbf{W}_l, \boldsymbol{b}_l\}_{l=1}^L =: \boldsymbol{\theta}_1$, $\{\boldsymbol{S}_l\}_{l=1}^L$, and history
       parameter $h$.
Output: The parameters of the trained learner.
1  $t \leftarrow 1$, $\boldsymbol{\delta} \leftarrow 0.1 \times \mathbf{1}$, $\boldsymbol{u}_{t-1} \leftarrow \mathbf{0}$          // initialization
2  while stopping condition is not met do
3      Compute $\ell_{\text{value}} = \ell_\tau(\boldsymbol{g}(\cdot \mid \boldsymbol{\theta}_t))$ and $\boldsymbol{u}_t = \frac{\partial \ell_\tau}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_t)$ via BP.
4      $\boldsymbol{g} \leftarrow \boldsymbol{u}_t - \boldsymbol{u}_{t-1}$
5      Add $(\boldsymbol{\delta}, \boldsymbol{g})$ to the BFGS history as the newest BFGS pair.
6      if the number of pairs in the BFGS history is greater than $h$ then
7          remove the oldest pair from the BFGS history
8      Compute $\boldsymbol{d}$ via Algorithm 2 using the BFGS history, $\mathbf{C}_0 = \mathbf{I}$, and $\boldsymbol{u}_t$.
9      $\alpha \leftarrow 1$
10     while $\ell_\tau(\boldsymbol{g}(\cdot \mid \boldsymbol{\theta}_t + \alpha\, \boldsymbol{d})) \geqslant \ell_{\text{value}} + 10^{-4} \alpha\, \boldsymbol{d}^\top \boldsymbol{u}_t$ do
11         $\alpha \leftarrow \alpha / 1.5$          // quasi-Newton line-search
12     $\boldsymbol{\delta} \leftarrow \alpha\, \boldsymbol{d}$
13     $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \boldsymbol{\delta}$
14     $t \leftarrow t + 1$
15 return $\boldsymbol{\theta}_t$ as the minimizer of the training loss
```

Adaptive Gradient Methods

The limited-memory BFGS method determines a search direction using the recent history of previously computed gradients $\{\boldsymbol{u}_t\}$ and input parameters $\{\boldsymbol{\theta}_t\}$.

This is because the BFGS pairs $\{\boldsymbol{\delta}_t, \boldsymbol{g}_t\}$ can be easily constructed from the identities: $\boldsymbol{\delta}_t = \boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t$ and $\boldsymbol{g}_t = \boldsymbol{u}_{t+1} - \boldsymbol{u}_t$.

In other words, using only past gradient computations and with little extra computation, it is possible to infer some of the second-order information contained in the Hessian matrix of $\ell_{\boldsymbol{\tau}}(\boldsymbol{\theta})$.

In addition to the BFGS method, there are other ways in which we can exploit the history of past gradient computations.

Normal Approximation Method

One approach is to use the normal approximation method, in which the Hessian of ℓ_τ at $\boldsymbol{\theta}_t$ is approximated via

$$\widehat{\mathbf{H}}_t = \gamma\, \mathbf{I} + \frac{1}{h} \sum_{i=t-h+1}^{h} \boldsymbol{u}_i \boldsymbol{u}_i^\top, \tag{5}$$

where $\boldsymbol{u}_{t-h+1}, \ldots, \boldsymbol{u}_h$ are the h most recently computed gradients and γ is a tuning parameter (for example, $\gamma = 1/h$). The search direction is then given by

$$-\widehat{\mathbf{H}}_t^{-1} \boldsymbol{u}_t,$$

which can be computed quickly in $\mathcal{O}(h^2\, d)$ time.

AdaGrad Method

The Adaptive Gradient or AdaGrad method completely bypasses the need to invert a Hessian approximation, by storing the diagonal of $\widehat{\mathbf{H}}_t$ and using the search direction:

$$-\mathrm{diag}(\widehat{\mathbf{H}}_t)^{-1/2}\boldsymbol{u}_t.$$

We can avoid storing any of the gradient history by instead using the slightly different search direction (componentwise defined):

$$-\boldsymbol{u}_t \Big/ \sqrt{\boldsymbol{v}_t + \gamma \times \mathbf{1}},$$

where the vector $\boldsymbol{v}_t$ is updated recursively via

$$\boldsymbol{v}_t = \left(1 - \frac{1}{h}\right)\boldsymbol{v}_{t-1} + \frac{1}{h}\boldsymbol{u}_t \odot \boldsymbol{u}_t.$$

With this updating of $\boldsymbol{v}_t$, the difference between the vector $\boldsymbol{v}_t + \gamma \times \mathbf{1}$ and the diagonal of the Hessian $\widehat{\mathbf{H}}_t$ will be negligible.

Adam Method

A more sophisticated version of AdaGrad is the adaptive moment estimation or Adam method, in which we not only average the vectors $\{\boldsymbol{v}_t\}$, but also average the gradient vectors $\{\boldsymbol{u}_t\}$.

Algorithm 4: Updating of Search Direction at Iteration t via Adam

```
Input: $\boldsymbol{u}_t, \widehat{\boldsymbol{u}}_{t-1}, \boldsymbol{v}_{t-1}, \boldsymbol{\theta}_t$, and parameters $(\alpha, h_v, h_u)$, equal to,
       e.g., $(10^{-3}, 10^3, 10)$.
Output: $\widehat{\boldsymbol{u}}_t, \boldsymbol{v}_t, \boldsymbol{\theta}_{t+1}$.
1 $\widehat{\boldsymbol{u}}_t \leftarrow \left(1 - \frac{1}{h_u}\right) \widehat{\boldsymbol{u}}_{t-1} + \frac{1}{h_u} \boldsymbol{u}_t$
2 $\boldsymbol{v}_t \leftarrow \left(1 - \frac{1}{h_v}\right) \boldsymbol{v}_{t-1} + \frac{1}{h_v} \boldsymbol{u}_t \odot \boldsymbol{u}_t$
3 $\widehat{\boldsymbol{u}}_t \leftarrow \widehat{\boldsymbol{u}}_t \Big/ \left(1 - (1 - h_u^{-1})^t\right)$
4 $\boldsymbol{v}_t \leftarrow \boldsymbol{v}_t \Big/ \left(1 - (1 - h_v^{-1})^t\right)$
5 $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \alpha \, \widehat{\boldsymbol{u}}_t \Big/ \left(\sqrt{\boldsymbol{v}_t} + 10^{-8} \times \mathbf{1}\right)$
6 return $\widehat{\boldsymbol{u}}_t, \boldsymbol{v}_t, \boldsymbol{\theta}_{t+1}$
```

Momentum Method

Yet another computationally cheap approach is the momentum method, in which the steepest descent iteration (1) is modified to

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha_t \, \boldsymbol{u}_t + \gamma \, \boldsymbol{\delta}_{t-1},$$

where $\boldsymbol{\delta}_{t-1} = \boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}$ and γ is a tuning parameter.

This strategy frequently performs better than the “vanilla” steepest descent method, because the search direction is less likely to change abruptly.

The stochastic gradient descent method, the limited-memory BFGS Algorithm 3, or any of the adaptive gradient methods in this section, can frequently handle networks with many hidden layers (provided that any tuning parameters and initialization values are carefully chosen via experimentation).