15 Improving First and Second-Order Methods by Modeling Uncertainty

Nicolas Le Roux nicolas.le.roux@gmail.com
Microsoft Research Cambridge
Yoshua Bengio yoshua.bengio@umontreal.ca
University of Montreal
Andrew Fitzgibbon awf@microsoft.com
Microsoft Research Cambridge

Machine learning's goal is to provide algorithms able to deal with new situations or data. Thus, what matters is their performance on unseen test data. For that purpose, we have at our disposal data on which we can train our model. Much previous work aimed at taking account of the fact that the training data are only a sample from the distribution of interest, for instance, by optimizing training error plus a regularization term. We present here a new way to take that information into account, based on an estimator of the gradient of generalization error that takes this uncertainty into account through a weak prior. We show how taking into account the uncertainty across training data can yield faster and more stable convergence, even when using a first-order method. We then show that in spite of apparent similarities with second-order methods, taking this uncertainty into account is different and can be used in conjunction with an approximate Newton method to yield even faster convergence.

15.1 Introduction

Machine learning often looks like optimization: write down the likelihood of some training data under some model and find the model parameters which

maximize that likelihood, or which minimize some divergence between the model and the data. In this context, conventional wisdom is that one should find in the optimization literature the state-of-the-art optimizer for one's problem, and use it.

However, this should not hide the fundamental difference between these two concepts: while optimization is about minimizing some error on the training data, it is the performance on the test data we care about in machine learning. Of course, by their very definition, test data are not available to us at training time, and thus we must use alternative techniques to prevent the model from focusing too much on the training data at the expense of generalization performance (a phenomenon known as overfitting), such as weight decay or limited model capacity.

The goal of this chapter is to prove that this misfit between training and test error can be dealt with by modifying the optimization procedure rather than the objective function itself, using a technique similar to the natural gradient. It is organized as follows: we start by exploring the differences between the optimization and the learning frameworks in section 15.2, before introducing a model of the gradients which modifies the search direction in section 15.3. Then, after exploring the similarities and differences between the covariance and the Hessian in section 15.4, we propose a modification of our model of the gradients, enabling us to use second-order information to find the optimal search direction, in section 15.5. Section 15.6 presents TONGA, an efficient algorithm to obtain these new search directions, which is tested in section 15.7.

15.2 Optimization Versus Learning

15.2.1 Optimization Methods

The goal of optimization is to minimize a function f, which we will assume to be twice differentiable and defined from a space E to $\mathbb{R}$, over E. This is a problem with a considerable literature (see Nocedal and Wright (2006), for instance). It is well known that second-order descent methods, which rely on the Hessian of f (or approximations thereof), enjoy much faster theoretical convergence than first-order methods (quadratic versus linear), in terms of number of updates. Such methods include the following: Newton, Gauss-Newton, Levenberg-Marquardt, and quasi-Newton (such as BFGS).

15.2.2 Online Learning

The learning framework for online learning differs slightly from the optimization one. The function f we wish to minimize (which we call the cost function) is defined as the expected value of a function $\mathcal{L}$ under a distribution p over the space E of possible inputs, that is,

$$f(\theta) = \int_{x \in E} \mathcal{L}(\theta, x) p(x)\, dx, \tag{15.1}$$

and we have access only to samples x_i drawn from p. If we have n samples, we can define a new function,

$$\widehat{f}(\theta) = \frac{1}{n} \sum_i \mathcal{L}(\theta, x_i)\ . \tag{15.2}$$

Let us call f the *test cost*, and $\widehat{f}$ the *training cost*. The x_i are the *training data*. As n goes to infinity, the difference between f and $\widehat{f}$ vanishes.

Bottou and Bousquet (2008) study the case where one has access to a potentially infinite amount of training data but only a finite amount of time. This setting, which they dub *large-scale learning*, calls for a tradeoff between the quality of the optimization for each data point and the number of data points treated. They show that

1. good optimization algorithms may be poor learning algorithms;
2. stochastic gradient descent enjoys a faster convergence rate than batch gradient descent;
3. introducing second-order information can win us a constant factor (the condition parameter).

Therefore, the choice lies between first- and second-order stochastic gradient descent, depending on the additional cost of taking second-order information into account and the condition parameter. Several authors have developed algorithms allowing for efficient use of this second-order information in a stochastic setting (Schraudolph et al., 2007; Bordes et al., 2009). However, we argue, all of these methods are derived from optimization methods without taking into account the particular nature of the learning problem.

More precisely, the gradient we would like to compute is the one of the true cost defined in (15.1). Differentiating both sides of this equation with respect to θ yields (assuming we can swap the integral and the derivative)

$$g^*(\theta_0) = \frac{\partial f}{\partial \theta}(\theta_0) = \int_{x \in E} \frac{\partial \mathcal{L}}{\partial \theta}(\theta_0, x_i) p(x)\, dx\ . \tag{15.3}$$

Similarly, differentiating both sides of (15.2) with respect to θ yields

$$g(\theta_0) = \frac{\partial \widehat{f}}{\partial \theta}(\theta_0) = \frac{1}{n} \sum_i \frac{\partial \mathcal{L}}{\partial \theta}(\theta_0, x_i) \; . \tag{15.4}$$

Thus, for each parameter value θ_0, the true gradient g^* is the expectation of $\frac{\partial \mathcal{L}}{\partial \theta}(\theta_0, x)$ under p, and we are given only samples $g_i = \frac{\partial \mathcal{L}}{\partial \theta}(\theta_0, x_i)$ from this distribution. This bears a lot of resemblance to the standard setting of machine learning: given a set of samples (here, the g_i's) from a distribution, one wishes to estimate interesting properties of that distribution (here, its expectation). We will thus proceed in the standard way, that is, we shall build a model of the gradients g_i and estimate its parameters. At this point, it is important to recall that our model is valid only for one value of θ_0, and thus needs to be reevaluated every time we move in parameter space. We shall discuss this issue further in section 15.3.1.

Also, note that the same reasoning may be applied to stochastic optimization. Modeling the distribution of the gradients will prevent us from focusing too much on the previously seen examples, which should enhance the final performance of the algorithm. This intuition will be proved in section 15.7.

15.3 Building a Model of the Gradients

We shall now describe in detail the model of gradients we use. Since our goal is to achieve fast treatment of incoming data, it must be simple enough to be estimated accurately with little computation. Additionally, a simpler model will regularize our estimate, making it more robust. We will thus use a Gaussian model, which we will see has the extra advantage of having an interpretation as an approximation to the central-limit theorem.

The only quantity we are interested in is the mean of this Gaussian, which is the true gradient g^*.

The likelihood term is

$$g_i | g^* \sim \mathcal{N}(g^*, C^*) \tag{15.5}$$

where C^* is the true covariance of the gradients, that is,

$$C^* = \int_x \left(\frac{\partial \mathcal{L}(\theta, x)}{\partial \theta} - g^* \right) \left(\frac{\partial \mathcal{L}(\theta, x)}{\partial \theta} - g^* \right)^T p(x) \, dx. \tag{15.6}$$

Indeed, according to the central-limit theorem, if the g_i's were averages of gradients over a minibatch, then their distribution would converge to a Gaussian as the minibatch size grows to infinity, thus yielding the correct

likelihood. Of course, one must bear in mind that this becomes an approximation for finite sizes, and even more so when g_i is a single gradient.

Before receiving any gradient, we know neither the direction nor the amplitude of the true gradient. Hence it is reasonable to take a prior over g^* that is centered on 0 and has an isotropic covariance:

$$g^* \sim \mathcal{N}(0, \sigma^2 I) \ . \tag{15.7}$$

Assuming we receive n gradients $g_1, \ldots, g_n$ with average g, the posterior distribution over g^* is obtained by combining (15.7) and (15.5), yielding

$$g^*|(g_1, \ldots, g_n) \sim \mathcal{N}\left(\left[I + \frac{C^*}{n\sigma^2}\right]^{-1} g, \left[nC^{*-1} + \frac{I}{\sigma^2}\right]^{-1}\right) \ . \tag{15.8}$$

Even though we care about only the mean g^*, its posterior depends on the covariance matrix C^*, which should be estimated using data. Though the proper Bayesian way would be to place a prior on C^* (which could be inverse-Wishart to keep the model conjugate) and estimate the joint posterior over (g, C^*), we will set C^* to the empirical covariance matrix C of the gradients. In doing so, we will lose in robustness but gain in computational efficiency.

Replacing C^* with the empirical covariance C in (15.8), we have the estimator

$$g^*|(g_1, \ldots, g_n) \sim \mathcal{N}\left(\left[I + \frac{C}{n\sigma^2}\right]^{-1} g, \left[nC^{-1} + \frac{I}{\sigma^2}\right]^{-1}\right) , \tag{15.9}$$

with

$$C = \frac{1}{n}\sum_{i=1}^{n}(g_i - g)(g_i - g)^T \ . \tag{15.10}$$

Now that we have estimated the posterior distribution over g^*, we can estimate the expected decrease in $\mathcal{L}$ for a given update $\Delta\theta$, which is simply

$$E[\Delta\mathcal{L}] = (\Delta\theta)^T E[g^*] \ . \tag{15.11}$$

For a given norm of $\Delta\theta$, the optimal decrease is obtained for $\Delta\theta \propto -E[g^*]$, that is,

$$(\Delta\theta)_{\text{opt}} \propto -\left[I + \frac{C}{n\sigma^2}\right]^{-1} g \ . \tag{15.12}$$

This quantity, which we call *consensus gradient*, is reminiscent of the natural gradient of Amari (1998). In his work, Amari showed that in a neural network, the direction of steepest descent in the Riemannian manifold

defined by that network is

$$(\Delta\theta)_{\text{Amari}} \propto -\left[GG^T + \lambda I\right]^{-1} g \;, \tag{15.13}$$

where G is the matrix containing one gradient per column (λI acts as a regularizer and the direction of steepest descent uses $\lambda = 0$). However, there are some important differences. First, here the covariance matrix C is centered and scaled, and thus not the Fisher information matrix, GG^T, as in Amari's work. Second, and perhaps more important, this formulation makes it obvious that the term $\frac{C}{n\sigma^2}$ acts here as a regularizer of the standard gradient direction, rather than defining a completely new direction based on another metric.

Let us pause a bit and analyze the behavior of such a direction. If there is a strong disagreement between gradients along a direction $\boldsymbol{d}$, the covariance will be large along this direction (that is, the value of $\boldsymbol{d}^T C \boldsymbol{d}$ will be large), which will reduce the update $\Delta\theta$ along $\boldsymbol{d}$. Thus, (15.12) naturally and gracefully deals with incoherent or noisy data. This is in stark contrast with, for example, outlier detectors which discard data points entirely. Moreover, the direction along which to shrink the updates is also learned and does not require any heuristic.

If, on the other hand, there is very little disagreement along a direction, then the step along this direction will be taken as usual. It is worth emphasizing that in this setting, the smallest eigenvalues of C are unimportant, as they will have very little effect on the final direction, as opposed to existing natural gradient methods, where they dominate the final update. As they are often harder to estimate correctly, those methods need to add a regularizer, which is unnecessary here. Once again, in our case the matrix C is the regularizer, not the identity matrix. soit clair. Le style me parat aussi lourd et redondant.

Figure 15.1 shows an example of consensus gradient and one of mean gradient directions, with varying amounts of disagreement among gradients.

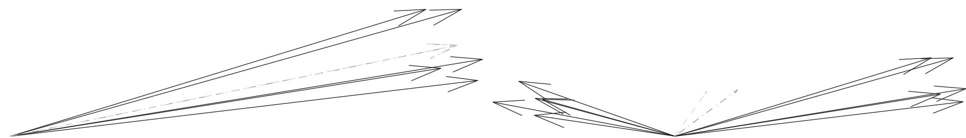

Figure 15.1: **Left**: when gradients (solid lines) agree, the consensus gradient direction (dashed line) is indistinguishable from the mean gradient (dashed-dotted line). **Right**: when gradients disagree, the consensus gradient shrinks in direction of high variance while leaving the others untouched.

The dot product of the empirical gradient and the consensus gradient direction of (15.12) is $g^T \left[I + \frac{C}{n\sigma^2}\right]^{-1} g$, which is always positive since $\left(I + \frac{C}{n\sigma^2}\right)$ is a positive definite matrix. Thus, though the consensus gradient direction is a modification of the original direction, they will never be in disagreement (that is, if the magnitude of the update is small enough, the training cost is guaranteed to decrease as well).

Moreover, when the number n of gradients goes to infinity, the optimal direction converges to g: this makes sense, since the modification to standard gradient descent arises from the uncertainty around the particular set of samples chosen, which becomes nonexistent in the case of infinite sample size. However, as we recover a standard optimization problem, we may be disappointed by the use of a first-order method, which, as mentioned earlier, is theoretically slower than second-order ones.

15.3.1 Setting a Zero-Centered Prior at Each Timestep

Before moving on to the second-order version of our consensus gradient algorithm, we will briefly comment on the choice of our prior at each step. Indeed, (15.9) has been obtained using the zero-centered Gaussian prior defined in (15.7). Except for the first update, one may wonder why we would use such a distribution rather than the posterior distribution at the previous timestep as our prior. There are two reasons for that. The first one is that whenever we update the parameters of our model, the distribution over the gradients changes. If the function to optimize were truly quadratic, we could quantify the change in gradient exactly using the Hessian. Unfortunately, this is not the case, and if this path is explored (as we believe it should be), it will involve approximations of the posterior. The second reason is computational. Even if we were able to follow the mean of the posterior exactly, the resulting distributions would become more and more complex over time (while still being Gaussians, their means and covariances would depend on a sum of covariance matrices). Thus, while acknowledging that using the prior of (15.7) at every timestep is a suboptimal strategy that future work might enhance, we believe that it is very appealing because of the simplicity of the algorithm.

15.4 The Relative Roles of the Covariance and the Hessian

In its original formulation, the natural gradient algorithm has often been considered as approximation to the Newton method. Indeed, their updates

look very similar ($\boldsymbol{d} = (GG^T)^{-1}g$ for the natural gradient and $\boldsymbol{d} = H^{-1}g$ for the Newton method), and there are several reasons to believe that the covariance (either centered, that is, C, or uncentered, that is, GG^T) and H have analogous properties. However, as we will see, they encode completely different kinds of information. From there, it seems natural to exploit both, yielding an algorithm combining their advantages.

15.4.1 Similarities Between C and H

Let us first focus on the similarities between the covariance matrix and the Hessian.

15.4.1.1 Maximum Likelihood

Let us assume that we are training a density model by minimizing the negative log-likelihood. The cost function f_{nll} is defined by

$$f_{\text{nll}}(\theta) = -\int_x \log[L(\theta, x)]p(x)\, dx \,. \tag{15.14}$$

Note that this L is related to the $\mathcal{L}$ used in section 15.2.2 through $\mathcal{L} = -\log L$, but with the constraint that L is a distribution. Let us consider the case where there is a parameter vector θ *such that our model is perfect* (where $p(x) = L(\theta, x)$) and that we are at this θ. Then the covariance matrix of the gradients at that point is equal to the Hessian of f_{nll}. In the general case, this equality does not hold.

15.4.1.2 Gauss-Newton

Gauss-Newton is an approximation to the Newton method when f can be written as a sum of squared residuals:

$$f(\theta) = \frac{1}{2}\sum_i f_i(\theta)^2. \tag{15.15}$$

Computing the Hessian of f yields

$$\frac{\partial^2 f(\theta)}{\partial \theta^2} = \sum_i f_i(\theta)\frac{\partial^2 f_i}{\partial \theta^2} + \sum_i \frac{\partial f_i}{\partial \theta}\frac{\partial f_i}{\partial \theta}^T \,. \tag{15.16}$$

If the f_i get close to 0 (relative to their gradient), the first term may be ignored, yielding the following approximation to the Hessian:

$$H \approx \sum_i \frac{\partial f_i}{\partial \theta}\frac{\partial f_i}{\partial \theta}^T \,. \tag{15.17}$$

One, however, must be aware of the following:

- this approximation is interesting only when the f_i are *residuals* (that is, when the approximation is valid close to the optimum);
- the gradients involved are those of f_i and not of f_i^2;
- the term on the right-hand side is the *uncentered* covariance of these gradients.

In order to compare the result of (15.17) to the natural gradient, we will assume that the sum in (15.15) is over data points, that is,

$$f(\theta) = \frac{1}{2}\sum_i f_i(\theta)^2 = \frac{1}{N}\sum_i \mathcal{L}(\theta, x_i) \tag{15.18}$$

with the cost for each data point being

$$\mathcal{L}(\theta, x_i) = \frac{N}{2} f_i(\theta)^2 \ . \tag{15.19}$$

The gradient of this cost with respect to θ is

$$g_i = \frac{\partial \mathcal{L}(\theta, x_i)}{\partial \theta} = N f_i(\theta)\frac{\partial f_i(\theta)}{\partial \theta} \ . \tag{15.20}$$

At the optimum (where the average of the gradients is zero and the centered and uncentered covariance matrices are equal), the covariance matrix of the g_i's is

$$C \ = \ \sum_i g_i g_i^T = N^2 \sum_i f_i(\theta)^2 \frac{\partial f_i}{\partial \theta}\frac{\partial f_i}{\partial \theta}^T \ , \tag{15.21}$$

which is a weighted sum of the terms involved in (15.17). Thus the natural gradient and the Gauss-Newton approximation, while related, are different quantities and (as we will show) have very different properties.

15.4.2 Differences Between C and H

Remember what the Hessian is: a measure of the change in gradient when we move in *parameter space*. In other words, the Hessian helps to answer the following question: *If I were at a slightly different position in parameter space, how different would the gradient be?* It is a quantity defined for any (twice differentiable) function.

On the other hand, the covariance matrix of the gradients captures the uncertainty around this particular choice of training data, that is, the change in gradient when we move in *input space*. In other words, the covariance helps us to answer the following question: *If I had slightly different training data,*

how different would the gradient be? This quantity makes only sense when there are training data.

Whereas the Hessian seems naturally suited to optimization problems (it allows us to be less shortsighted when minimizing a function), the covariance matrix unleashes its power in the learning setting, where we are given only a subset of the data. We do not claim that there are no numerical similarities between them, and indeed the experiments hint at differing and complementary effects, so we really wish to clarify how they differ.

From this observation, it seems natural to combine these two matrices.

15.5 A Second-Order Model of the Gradients

In our first model of the gradients, in section 15.3, we did not assume any particular form of the function $\mathcal{L}$ to minimize. In the Newton method, however, one assumes that the cost function is locally quadratic, that is,

$$\mathcal{L}(\theta) \approx f(\theta) = \frac{1}{2}(\theta - \theta^*)^T H(\theta - \theta^*) \tag{15.22}$$

for some value of θ^*.

The derivative of this cost is

$$g^*(\theta) = \frac{\partial f(\theta)}{\partial \theta} = H(\theta - \theta^*) \ . \tag{15.23}$$

We can see that in the context of a quadratic function, the isotropic prior over g^* proposed in (15.7) is erroneous, as g^* is clearly influenced by H. We shall, rather, consider an isotropic Gaussian prior on the quantity $\theta - \theta^*$, as we do not have any information about the position of θ relative to θ^*. The resulting prior distribution over g^* is

$$g^* \sim \mathcal{N}\left(0, \sigma^2 H^2\right) , \tag{15.24}$$

where we omit the dependence on θ to keep the notation uncluttered. In a fashion similar to section 15.3, we will suppose that we are given only a finite training set composed of n data points x_i with associated gradients g_i. The empirical gradient g is the mean of the g_i's. Using the central limit theorem, we again have

$$g|g^* \sim \mathcal{N}\left(g, \frac{C^*}{n}\right) \tag{15.25}$$

where C^* is the true covariance of the gradients, which we will once again replace with the empirical covariance C. Therefore, the posterior distribution

over g is

$$g^*|g \sim \mathcal{N}\left(\left[I + \frac{CH^{-2}}{n\sigma^2}\right] g, \left[\frac{H^{-2}}{\sigma^2} + nC^{-1}\right]^{-1}\right) . \quad (15.26)$$

Since the function $\mathcal{L}$ is locally quadratic, we wish to move in the direction $H^{-1}g$. This direction follows the Gaussian distribution

$$H^{-1}g|g \sim \mathcal{N}\left(\left[I + \frac{H^{-1}CH^{-1}}{n\sigma^2}\right]^{-1} H^{-1}g, \left[\frac{I}{\sigma^2} + nHC^{-1}H\right]^{-1}\right) . \quad (15.27)$$

Since the mean of the Gaussian in (15.27) appears complicated, we shall explain it. Let us write d_i for the Newton directions:

$$d_i = H^{-1}g_i . \quad (15.28)$$

Since C is the covariance matrix of the gradients g_i, $H^{-1}CH^{-1} = C^H$ is the covariance matrix of the d_i's. We can therefore rewrite

$$H^{-1}g|g \sim \mathcal{N}\left(\left[I + \frac{C^H}{n\sigma^2}\right]^{-1} d, \left[\frac{I}{\sigma^2} + n(C^H)^{-1}\right]^{-1}\right) \quad (15.29)$$

where d is the average of the Newton directions, that is, $d = H^{-1}g$. The direction which maximizes the expected gain is thus

$$\Delta\theta \propto -\left[I + \frac{C^H}{n\sigma^2}\right]^{-1} d . \quad (15.30)$$

This formula is exactly the consensus gradient (15.12), but on the Newton directions. This makes perfect sense, as the Newton method is the standard gradient descent on a space linearly reparameterized by $H^{0.5}$. Here, the direction is the one obtained after having computed the consensus gradient in the same linearly reparameterized space.

From a computational perspective, this simple combination is excellent news. It means that one may choose his or her favorite second-order gradient descent method to compute the Newton directions, and then his or her favorite consensus gradient algorithm to apply to these Newton directions, to yield an algorithm combining the advantages of both methods.

As a side note, one can see that as the number n of data points used to compute the mean increases, the prior vanishes and the posterior distribution concentrates around the empirical Newton direction. This is in contrast with the method of section 15.3, which converged to the first-order gradient descent algorithm.

15.6 An Efficient Implementation of Online Consensus Gradient: TONGA

So far, we have

- provided a justification for the consensus gradient as a means of dealing with the uncertainty arising from having only a finite number of samples in our dataset;
- explored the similarities and differences between the covariance matrix C and the Hessian H;
- shown how the information in these two matrices could be combined to yield an efficient algorithm, both from an optimization and from a learning point of view.

However, these techniques require matrix inversions, which makes them unsuitable for practical cases, where the number of model parameters and of training data may be very large. Also, since our main focus is online learning, we wish to be able to update our parameters after each example (stochastic), or each small group of examples (minibatch), as recommended in Bottou and Bousquet (2008).

Section 15.6.1 will uncover a set of optimizations and approximations which renders possible fast online natural or consensus gradient algorithms: TONGA. This algorithm will provide the basis for the second-order version using the Hessian.

15.6.1 Computing a Low-Rank Approximation of the Covariance Matrix

In a model with P parameters, the covariance C of the gradients over n data points takes $O(nP^2)$ to compute and has an $O(P^2)$ memory storage requirement. Computing its first k eigenvectors is in $O(kP^2)$. When P is large, none of these operations is feasible. This section will thus introduce a way of finding the first k eigenvectors of the covariance matrix without ever storing it.

For the moment, we will assume that the centered covariance matrix may be written in the form $C = GG^T$ for some matrix G. The proof that such a factorization is possible and the explicit formula for G will appear in section 15.6.2. We will assume that G has n columns (and P rows for C to have the correct size).

Writing G in terms of its compact SVD, we get

$$G = U_G \Sigma_G V_G^T, \tag{15.31}$$

where (assuming we have $n < P$) U_G is of size $P \times n$, and Σ_G and V_G are of size $n \times n$. With this notation, the eigenvectors of C associated with non-zero eigenvalues are the columns of U_G and the associated eigenvalues are the diagonal elements of Σ_G^2.

Let us now consider the matrix

$$D = G^T G \ . \tag{15.32}$$

This is an $n \times n$ matrix whose eigenvectors are the columns of V_G and whose eigenvalues are the same as those of C. Left-multiplying those eigenvectors by G and right-multiplying them by Σ_V^{-1}, we get

$$G V_G \Sigma_G^{-1} = U_G \ . \tag{15.33}$$

Thus, we can retrieve the first k eigenvectors and eigenvalues of C by computing D (for a cost of $O(Pn^2)$), extracting its first k eigenvectors (for a cost of $O(kn^2)$), and then performing the matrix multiplications (for a cost of $O(Pn^2)$). Therefore, if n is much smaller than P, this method is much faster than computing C and its eigenvectors directly ($O(Pn^2)$, instead of $O(nP^2)$).

Another advantage is that it is never required to store or even compute C, but only to have access to the matrix G. Section 15.6.2 will show how to get this matrix G efficiently whenever a new data point comes in.

15.6.2 A Fast Update of the Covariance Matrix

Since efficiency is our main goal, we need a fast way to update the covariance matrix of the data points as they arrive. Also, we need to satisfy two constraints. First, the covariance needs to be estimated over many data points. Second, as it will change during the optimization, we need to progressively reduce the contribution of the older data points and replace it with the contribution of the newer ones. For that purpose, we shall use exponentially moving mean μ_n and covariance C_n:

$$\mu_1 = g_1 \tag{15.34}$$

$$C_1 = 0 \tag{15.35}$$

$$\mu_n = \gamma \mu_{n-1} + (1-\gamma) g_n \tag{15.36}$$

$$C_n = \gamma C_{n-1} + \gamma(1-\gamma)(g_n - \mu_{n-1})(g_n - \mu_{n-1})^T \tag{15.37}$$

where g_i is the gradient obtained at time step i and γ is the discount factor. The closer γ is to 1, the longer an example seen at time t will influence the means and covariance estimated at later times.

Thus, since we wish to keep a factorization of C under the form GG^T,

whenever a new gradient g_n comes in, we simply have to

1. multiply G by $\sqrt{\gamma}$
2. append the column $(g_n - \mu_{n-1})\sqrt{\gamma(1-\gamma)}$ to G.

15.6.3 Finding the Consensus Gradient Direction Between Two Updates

In section 15.6.1, we have showed that computing the first k eigenvectors of the matrix $C = GG^T$ when G has n columns and is in $O(Pn^2)$. We could thus use the following strategy:

1. compute the first k eigenvectors of C,
2. compute the consensus gradient update using the eigendecomposition of C,
3. write the low-rank approximation under the form UU^T (U then being the matrix of unnormalized eigenvectors),
4. update the matrix U when a new data point arrives, following section 15.6.2, where G plays the role of U,
5. recompute the first k eigenvectors of the new C for a cost of $O(P(k+1)^2)$,
6. iterate from 2.

One can see that the cost of this algorithm is $O(Pk^2)$ for every new gradient, which is approximately k^2 slower than standard gradient descent. The idea will thus be to update this covariance matrix as new data points arrive, but not to recompute the eigendecomposition every time. Instead, we will add data points until there are $k + B$ vectors in the matrix G (with B a hyperparameter), at which point we will recompute the eigendecomposition of this new covariance matrix.

There is a problem, however. While it is easy to compute the consensus gradient direction when one has access to the eigendecomposition of C, this will not be the case when several data points have been added. Luckily, the computation remains tractable, as we will see. b steps after the last eigendecomposition, the matrix G may be written as

$$G = [K_0 U \quad K_1(g_1 - \mu_0) \dots K_b(g_b - \mu_{b-1})]$$

(the constants $K_0, \dots, K_b$ stem from the $\sqrt{\gamma}$ and $\sqrt{\gamma(1-\gamma)}$ factors of section 15.6.2). Since $C = GG^T$, and in order to compute the naturalized gradient $\boldsymbol{d} = (I + C/[n\sigma^2])^{-1} g_b$, we wish to find the direction $\boldsymbol{d}$ such that

$$\left(I + \frac{GG^T}{n\sigma^2}\right)\boldsymbol{d} = g_b \ . \tag{15.38}$$

We will assume that $\boldsymbol{d}$ is of the form $\boldsymbol{d} = Gx + \lambda\mu_{b-1}$ for some vector x and some value of λ. With $y = [0\ \dots\ 0\ (1/K_b)]^T$, we have $g_b = Gy + \mu_{b-1}$, and (15.38) thus becomes

$$Gx + \lambda\mu_{b-1} + \frac{GG^T Gx + \lambda GG^T \mu_{b-1}}{n\sigma^2} = Gy + \mu_{b-1}\ .$$

Using $\lambda = 1$ and moving the fraction to the right-hand side, we get

$$Gx = Gy - \frac{GG^T Gx + \lambda GG^T \mu_{b-1}}{n\sigma^2} \tag{15.39}$$

$$x = \left(I + \frac{G^T G}{n\sigma^2}\right)^{-1}\left(y - \frac{G^T\mu_{b-1}}{n\sigma^2}\right) \tag{15.40}$$

(assuming G is of full rank), yielding

$$\boldsymbol{d} = G\left(I + \frac{G^T G}{n\sigma^2}\right)^{-1}\left(y - \frac{G^T\mu_{b-1}}{n\sigma^2}\right) + \mu_{b-1}\ . \tag{15.41}$$

Since G is of size $P \times (k+b)$, computing $\boldsymbol{d}$ costs $O((k+b)^3 + P(k+B)) = O(P(k+B))$, since we will limit ourselves to the setting where the rank of the covariance matrix is much less than the square root of the number of parameters.

15.6.4 Analysis of the Computational Cost

We will now briefly analyze the average computational cost of a gradient update where there are P parameters. We will assume that the gradients are computed over minibatches of size m:

1. every B steps, we compute the first k eigenvectors of the covariance matrix of $k+B$ data points for a total cost of $O(P(k+B)^2)$ (see section 15.6.1)
2. every step, we compute the consensus gradient direction for a total cost of $O(P(k+B))$ (see section 15.6.3)
3. computing the average gradient over a minibatch costs $O(Pm)$ at every step.

The average cost per update is thus $O\left(P\left[m + k + B + \frac{(k+B)^2}{B}\right]\right)$ as opposed to $O(Pm)$ for standard minibatch gradient descent. Thus, if we keep $(k+B)$ close to m, the cost of each iteration will be of the same order of magnitude as the standard gradient descent.

15.6.5 Block-Diagonal Online Consensus Gradient for Neural Networks

We now have a strategy to compute the consensus gradient direction using a low-rank approximation of the covariance matrix (with the rank varying between k and $k+B$). The question remains as to which value of k provides a reasonable approximation. Unfortunately, experiments showed that, in general, a high value of k (around 200 for $P = 2000$) was necessary for $\boldsymbol{d}$ to be a meaningful modification of the original gradient direction. In this setting, provided m, the minibatch size, is small (between 5 and 10), each update is at least 20 times slower than the standard gradient descent, and this extra computational cost cannot be made up by better search directions.

One might thus wonder if there are better approximations of the covariance matrix C than computing its first k eigenvectors. Collobert (2004) showed that the Hessian of a neural network with one hidden layer trained with the cross-entropy cost converges to a block-diagonal matrix during optimization. These blocks are composed of the weights linking all the hidden units to one output unit and all the input units to one hidden unit (*fan-in*). Since we listed some of the numerical similarities between the Hessian and the covariance, it may be useful to investigate the use of such a block structure for the covariance estimator. We will thus use a block-diagonal approximation of the covariance matrix. Instead of computing the first k eigenvectors of the entire covariance matrix, we will compute the first k eigenvectors of each block. Some remarks are worth making on that point:

- the rank of the approximation is not k but $k \times$ (number of blocks), which is much higher;
- all the terms outside of these blocks are set to 0. Thus, this approximation will be better only if these elements are actually negligible in the original covariance matrix;
- one may pick a different value of k for each block, depending on its size or the knowledge one has about the problem.

Figure 15.2 shows the correlation between the standard stochastic gradients of the parameters of a $16-50-26$ neural network. The first blocks represent the weights going from the input units to each hidden unit (thus 50 blocks of size 17, bias included), and the following blocks represent the weights going from the hidden units to each output unit (26 blocks of size 51). One can see that the block-diagonal approximation is reasonable. In the matrices shown in figure 15.2, which are of size 2176, a value of $k = 5$ yields an approximation of rank 380.

Another way of verifying the validity of our block-diagonal assumption

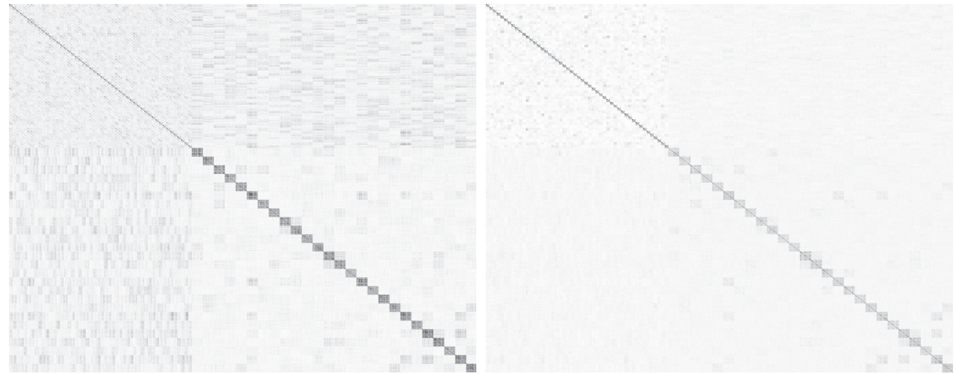

Figure 15.2: Absolute value of correlation between the standard stochastic gradients after one epoch in a neural network with 16 input units, 50 hidden units and 26 output units when following stochastic gradient directions (left) and consensus gradient directions (right). The first blocks in the diagonal are for input to hidden weights (per hidden unit), and the larger ones that follow are for hidden output weights (per output unit), showing a strong within-block correlation. One can see that the off-block terms are not zero, but still are much smaller than the terms in the block. Also, following natural directions helped in making the covariance more block-diagonal, though the reason behind it is unknown.

is to compute the error induced by our low-rank approximations, with or without this assumption. Figure 15.3 shows the relative approximation error of the covariance matrix as a ratio of Frobenius norms $\frac{\|C-\bar{C}\|_F^2}{\|C\|_F^2}$ for different types of approximations $\bar{C}$ (full or block-diagonal). We can first notice that approximating only the blocks yields a ratio of .35 (in comparison, taking only the diagonal of C yields a ratio of .80), even though we considered only $82,076$ out of the $4,734,976$ elements of the matrix (1.73 percent of the total). This ratio is almost obtained with $k = 6$. We can also notice that for $k < 30$, the block-diagonal approximation is much better (in terms of the Frobenius norm) than the full approximation, which proves its effectiveness in the case of neural networks. Yet this approximation also readily applies to any mixture algorithm where we can assume some form of decoupling between the components.

Thus in all our experiments, we used a value of $k = 5$, which allowed us to keep a cost per iteration of the same order of magnitude as standard gradient descent.

15.7 Experiments

In our experiments we wish to validate the two claims we have made so far:

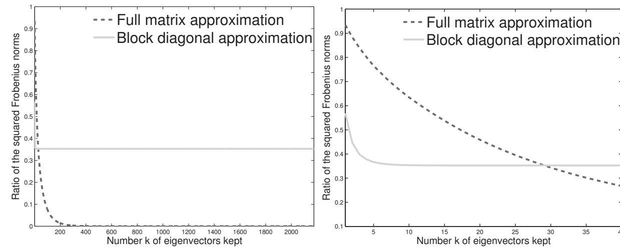

Figure 15.3: Quality of the approximation $\bar{C}$ of the covariance C, depending on the number of eigenvectors kept (k), in terms of the ratio of the Frobenius norms $\frac{\|C-\bar{C}\|_F^2}{\|C\|_F^2}$, for different types of approximation $\bar{C}$ (full matrix or block-diagonal). On the right we zoom on smaller values of k where the full matrix low-rank approximation overtakes the block-diagonal approximation.

1. that taking the uncertainty into account will speed up learning;
2. that C and H encode different pieces of information and that combining them will lead to even faster convergence.

The first set of experiments will thus compare TONGA with standard stochastic gradient descent, whereas the second will compare an approximate Newton method with the second-order TONGA, which we call *Natural-Newton*.

15.7.1 Datasets

Several datasets, architectures, and losses were used in our experiments.

15.7.1.1 Experiments with TONGA

We tried TONGA on two different datasets:

1. the MNIST digits dataset consists of 50,000 training samples, 10,000 validation samples, and 10,000 test samples, each one composed of 784 pixels. There are 10 classes (one for every digit)
2. the UCI USPS dataset consists of 9298 samples (broken into 6291 for training, 1000 for validation, and 2007 for the official test set), each one composed of 256 pixels. There are 10 different classes (one for every digit).

In both cases we minimized the negative log-likelihood on the training set, using a neural network with one hidden layer. The block-diagonal approximation of section 15.6.5 was used for TONGA, and no second-order information was used.

15.7.1.2 Experiments with Natural-Newton (Second-Order TONGA)

Whereas the goal of the experiments with TONGA was to determine if it was possible to use the information contained in the covariance matrix in an efficient manner, the experiments with Natural-Newton aim at exploring the differences between C and H.

As mentioned in section 15.5, one needs to use a second-order gradient descent method to compute the Newton directions. We chose to use the SGD-QN algorithm (Bordes et al., 2009), since it had recently won the Wild Track competition at the Pascal Large Scale Learning Challenge, on the same datasets it was used on: Alpha, Gamma, Delta, Epsilon, Zeta, and Face.

Labels were available only for the training examples of the challenge. We therefore split these examples into several sets:

- the first 100K (1M for the Face dataset) examples constituted our training set;
- the last 100K (1M for the Face dataset) examples constituted our test set.

The architecture used was a linear SVM. We did not change the hyperparameters of the SGD-QN algorithm; the interested reader may find them in the original paper.

Since this method uses a diagonal approximation to the Hessian, we decided to use a diagonal approximation to the covariance matrix. Though this was not required, and we could have used a low-rank covariance matrix, using a diagonal approximation shows the improvements over the original method that one can obtain with little extra effort. Thus, though (15.30) was used, none of the tricks presented in section 15.6 were necessary, except for the exponentially moving covariance matrix.

15.7.2 Experimental Details for Natural-Newton

15.7.2.1 Frequency of Updates

The covariance matrix of the gradients changes very slowly. Therefore, one does not need to update it as often as the Hessian approximation. In the SGD-QN algorithm, the authors introduce a counter *skip* which specifies how many gradient updates are done before the approximation to the Hessian is updated. We introduce an additional variable $skipC$, which specifies how many Hessian approximation updates are done before updating the covariance approximation. The total number of gradient updates between two covariance approximation updates is therefore $skip \cdot skipC$.

Experiments using the validation set showed that using values of *skipC* lower than 8 did not yield any improvement while increasing the cost of each update. We therefore used this value in all our experiments. This allows us to use the information contained in the covariance with very little computation overhead.

15.7.2.2 Limiting the Influence of the Covariance

Equation (15.29) tells us that the direction to follow is

$$\left[I + \frac{C^H}{n\sigma^2}\right]^{-1} \widehat{d}. \tag{15.42}$$

The only unknown in this formula is σ^2, which is the variance of our Gaussian prior on $\theta - \theta^*$. To avoid having to set this quantity by hand at every time step, we will devise a heuristic to find a sensible value of σ^2. While this will lack the purity of a full Bayesian treatment, it will allow us to reduce the number of parameters to be set by hand, which we think is a valuable feature of any gradient descent algorithm.

If we knew the distance from our position in parameter space, θ, to the optimal solution, θ^*, then the optimal value for σ^2 would be $\|\theta - \theta^*\|^2$. Of course, this information is not available to us. However, if the function to be optimized were truly quadratic, the squared norm of the Newton direction would be exactly $\|\theta - \theta^*\|^2$. We shall therefore replace σ^2 with the squared norm of the last-computed Newton direction. Since this estimate may be too noisy, we will replace it with the squared norm of the running average of the Newton directions, that is, $\|\mu_n\|^2$.

Even then, however, we may still get undesirable variations. We shall therefore adopt a conservative strategy: we will set an upper bound on the correction to the Newton method brought by (15.42). More precisely, we will bound the eigenvalues of $\frac{C^H}{n\|\mu_n\|^2}$ by a positive number B_C. The parameter update then becomes

$$\theta_n - \theta_{n-1} = -\left[I + \min\left(B_C, \frac{C^H}{n\|\mu_n\|^2}\right)\right]^{-1} H^{-1} g_n \tag{15.43}$$

where B_C is a scalar hyperparameter and $\min(B_C, M)$ is defined for symmetric matrices M with eigenvectors $u_1, \ldots, u_n$ and eigenvalues $\lambda_1, \ldots, \lambda_n$ as

$$\min(B_C, M) = \sum_{i=1}^{n} \min(B_C, \lambda_i) u_i u_i^T \tag{15.44}$$

(we bound each eigenvalue of M by B_C). If we set $B_C = 0$, we recover the standard Newton method. This modification transforms the algorithm in a conservative way, trading off potential gains brought by the covariance matrix for guarantees that the parameter update will not differ too much from the Newton direction.

In our experiments, the last 50K (500K for the Face dataset) examples of the training set were used as validation examples to tune the bound B_C defined in (15.43).

The pseudocode for the full algorithm, which we call *Natural-Newton*, is shown in algorithm 15.1.

Algorithm 15.1 Pseudocode of the Natural-Newton algorithm

Require: : skip (number of gradient updates between Hessian updates)
Require: : skipC (number of Hessian updates between covariance updates). Default value is skipC = 8.
Require: : θ_0 (the original set of parameters)
Require: : γ (the discount factor). Default value is 0.995.
Require: : T (the total number of epochs)
Require: : t_0
Require: : λ (the weight decay)
Require: : B_C the bound on the eigenvalues of the covariance matrix. Default value is $B_C = 2$.

1: t = 0, count = skip, countC = skipC
2: $H^- = \lambda\,\mathbf{I}$, $\mathbf{D} = \mathbf{I}$
3: $\mu_0 = 0$ (the running mean vector), $C_0^H = 0$ (the running covariance matrix)
4: **while** $t \neq T$ **do**
5: $\quad g_t \leftarrow \frac{\partial \mathcal{L}(\theta_t, x_t, y_t)}{\partial \theta_t}$
6: $\quad \theta_{t+1} \leftarrow \theta_t - (t + t_0)^{-1} \mathbf{D} H^- g_t$
7: $\quad$ **if** count == 0 **then**
8: $\quad\quad$ count ← skip
9: $\quad\quad$ Update H^-, the approximate inverse Hessian computed by SGD-QN.
10: $\quad\quad$ **if** countC == 0 **then**
11: $\quad\quad\quad$ countC ← skipC
12: $\quad\quad\quad \mu_t \leftarrow \gamma \mu_{t-1} + (1-\gamma) d_t$
13: $\quad\quad\quad C_t^H \leftarrow \gamma C_{t-1}^H + \gamma(1-\gamma)(d_t - \mu_{t-1})(d_t - \mu_{t-1})^T$
14: $\quad\quad\quad \mathbf{D} = \left(I + \frac{min(B_C, C_{t+1}^H)}{N \cdot \|\mu_{t+1}\|^2}\right)^{-1}$
15: $\quad\quad$ **else**
16: $\quad\quad\quad$ countC ← countC - 1
17: $\quad\quad$ **end if**
18: $\quad$ **else**
19: $\quad\quad$ count ← count - 1
20: $\quad$ **end if**
21: **end while**

15.7.2.3 Parameter Tuning

In all the experiments γ has been set to 0.995, as in TONGA. Again, to test the sensitivity of the algorithm to this parameter, we tried other values (0.999, 0.992, 0.99, and 0.9) without noticing any significant difference in validation errors.

We optimized the bound on the covariance (section 15.7.2.2) based on validation set error. The best value was chosen for the test set, but we found that a value of 2 yielded near-optimal results on all datasets, the difference between $B = 1$, $B = 2$, and $B = 5$ being minimal, as shown in figure 15.6 in the case of the Alpha dataset.

15.7.3 Results

15.7.3.1 TONGA

We performed a small number of experiments with TONGA's low-rank approximation of the full covariance matrix, keeping the overhead of the consensus gradient small (i.e., limiting the rank of the approximation). Regrettably, TONGA performed only as well as stochastic gradient descent, while being rather sensitive to the hyperparameter values. The following experiments, on the other hand, use TONGA with the block-diagonal approximation and yield impressive results. We believe this is a reflection of the phenomenon illustrated in figure 15.3 (right): the block-diagonal approximation makes for a very cost-effective approximation of the covariance matrix. All the experiments have been done by optimizing hyperparameters on a validation set (not shown here) and selecting the best set of hyperparameters for testing, trying to keep the overhead small due to natural gradient calculations.

One could worry about the number of hyperparameters of TONGA. However, default values of $k = 5$, $B = 50$, and $\gamma = .995$ yielded good results in every experiment.

Figure 15.4 shows that in terms of training CPU time (which includes the overhead due to TONGA), TONGA allows much faster convergence in training NLL, as well as in testing classification error and NLL than ordinary stochastic and minibatch gradient descent on this task. Also note that the minibatch stochastic gradient is able to profit from matrix-matrix multiplications, but this advantage is seen mainly in training classification error.

Note that the gain obtained on the USPS dataset is much slimmer. One possibility is that since the training set is much smaller, the independence

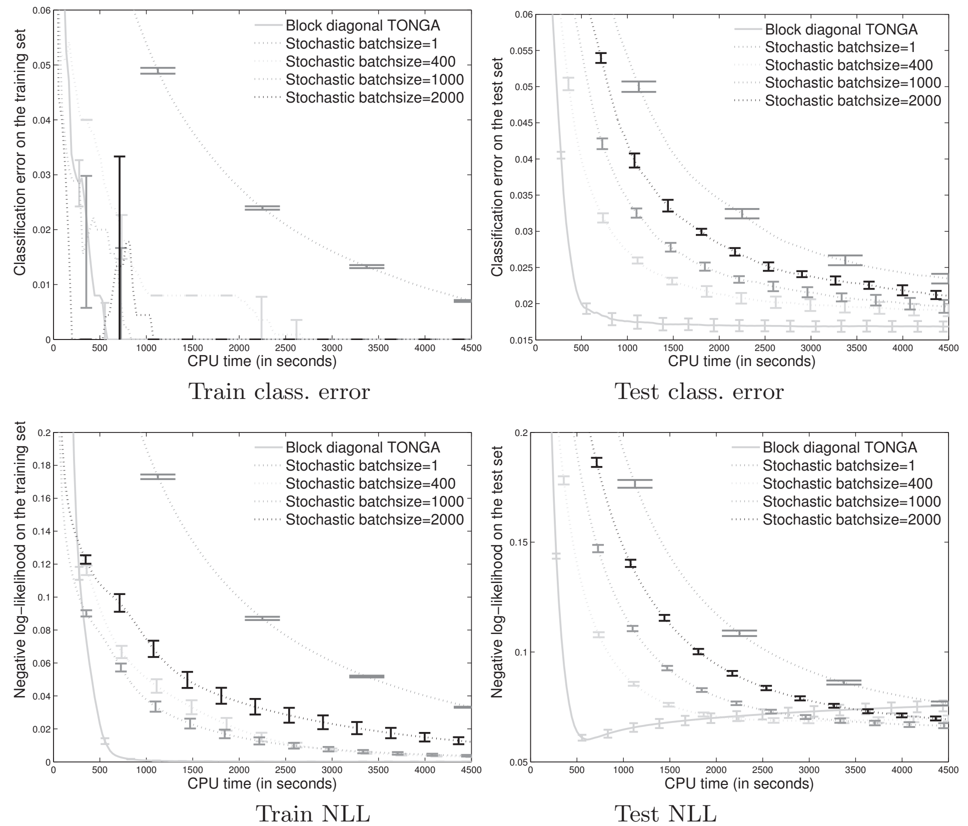

Figure 15.4: Comparison between stochastic gradient (with different minibatch sizes) and TONGA on the MNIST dataset, in terms of training (50,000 examples) and test (10,000 examples) classification error and negative log-likelihood (NLL). The mean and standard error have been computed using nine different initializations.

assumption used to obtain (15.9) becomes invalid.

Finally, though we expected an improvement only on the convergence speed of the test error, the training error decreased faster when using TONGA. This may be due to the stochastic nature of the optimization, where using the covariance prevented disagreeing gradients from having too much influence and ultimately slowing down the optimization.

15.7.3.2 Natural-Newton

Natural-Newton exhibited various behaviors on the datasets it was tried on:

- Natural-Newton never performs worse than SGD-QN and always better than TONGA. Using a large value of *skipC* ensures that the overhead of

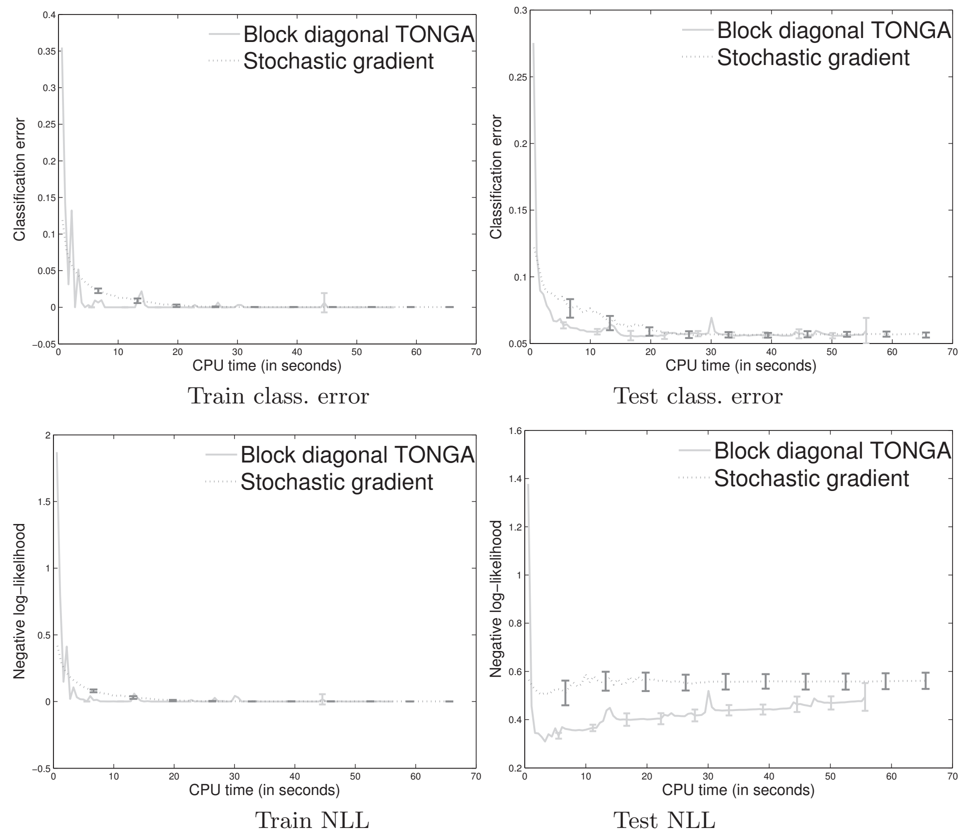

Figure 15.5: Comparison of stochastic gradient and TONGA on the USPS dataset, in terms of training (6291 examples) and test (2007 examples) classification error and negative log-likelihood (NLL). The mean and standard error were computed using nine different initializations.

using the covariance matrix is negligible.

- On the Alpha dataset, using the information contained in the covariance resulted in significantly faster convergence, with or without second-order information.
- On the Epsilon, Zeta, and Face datasets, using the covariance information stabilized the results while yielding the same convergence speed. This is in accordance with the use of the covariance, which reduces the influence of directions where gradients vary wildly.
- On the Gamma and the Delta datasets, using the covariance information helped a lot when the Hessian was not used, and provided no improvement otherwise.

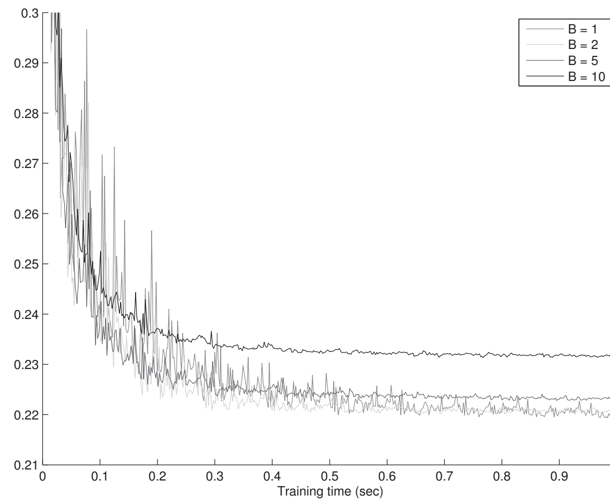

Figure 15.6: Validation error versus time on the Alpha dataset, for various values of B.

15.8 Conclusion

A lot of effort has been put into designing efficient online optimization algorithms, with great results. Most of these algorithms rely on some approximation to the Hessian or to the covariance matrix of the gradients. While the latter is commonly believed to be an approximation of the former, we showed that they encode very different kinds of information. Based on this, we proposed a way of combining information contained in the Hessian and in the covariance matrix of the gradients.

Experiments showed that on most datasets, our method offered either faster convergence or increased robustness, compared with the original algorithm. Furthermore, the second-order version of our algorithm never performed worse than the Newton algorithm it was built upon.

Moreover, our algorithm is able to use any existing second-order algorithm as base method. Therefore, while we used SGD-QN for our experiments, one may pick any algorithm best suited for a given task.

We hope to have shown two things. First, the covariance matrix of the gradients is usefully viewed not as an approximation to the Hessian, but as a source of additional information about the problem, for typical machine learning objective functions. Second, it is possible with little extra effort to use this information in addition to that provided by the Hessian matrix, in some cases yielding faster or more robust convergence.

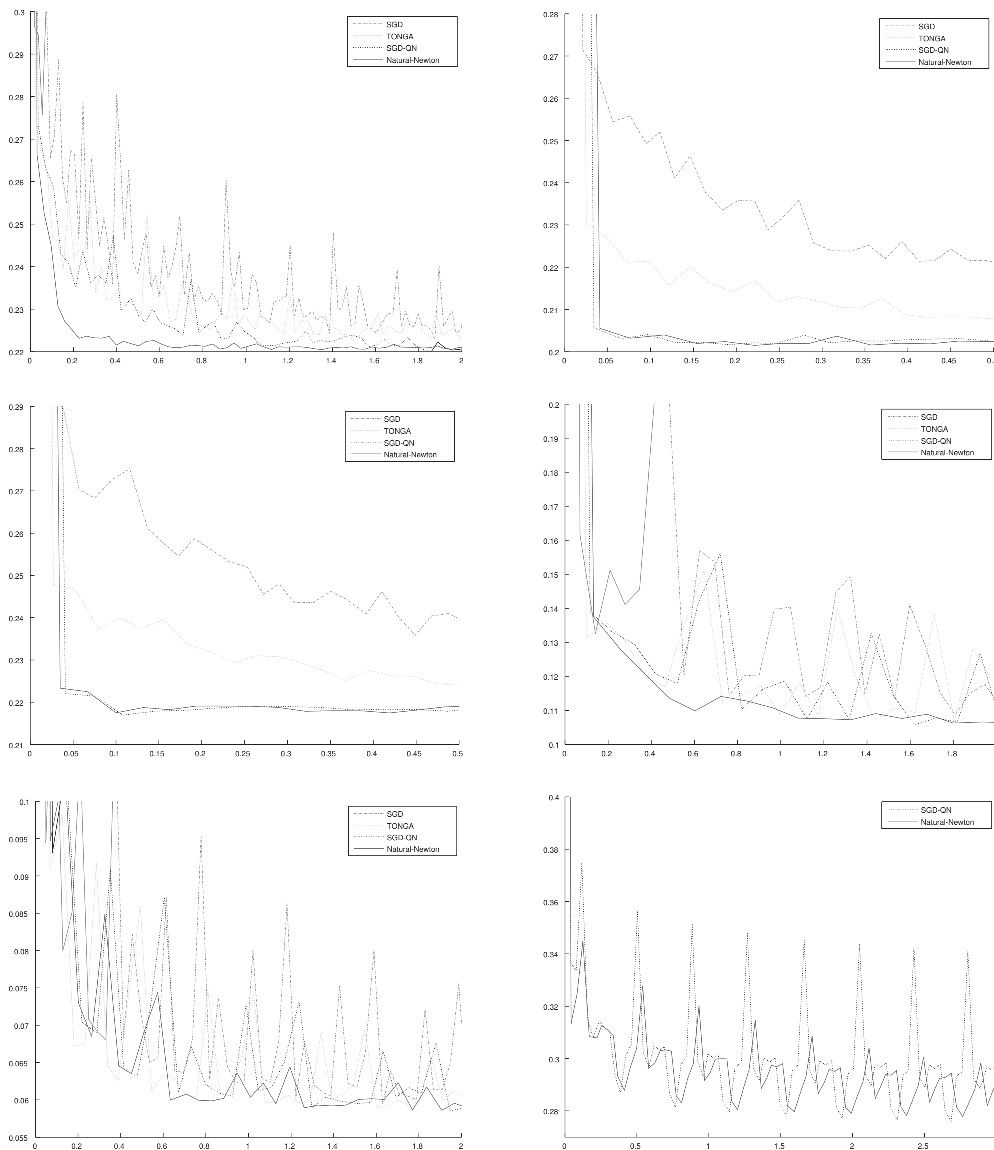

Figure 15.7: Test error versus time on the Alpha (top left), the Gamma (top right), the Delta (middle left), the Epsilon (middle right), the Zeta (bottom left), and the Face (bottom right) datasets.

Despite all these successes, we believe that these algorithms may be improved in several ways, whether it is by retaining some of the information contained in the posterior distribution between timesteps or in the selection of the parameter σ^2.

15.9 References

S. Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10 (2):251–276, 1998.

A. Bordes, L. Bottou, and P. Gallinari. SGD-QN: Careful quasi-newton stochastic gradient descent. *Journal of Machine Learning Research*, 10:1737–1754, July 2009.

L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In J. C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 161–168. MIT Press, 2008.

R. Collobert. *Large Scale Machine Learning*. PhD thesis, Université de Paris VI, LIP6, 2004.

J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Verlag, New York, second edition, 2006.

N. N. Schraudolph, J. Yu, and S. Günter. A stochastic quasi-Newton method for online convex optimization. In M. Meila and X. Shen, editors, *Proceedings of the 11th International Conference on Artificial Intelligence and Statistics*, volume 2 of *Workshop and Conference Proceedings*, pages 433–440. MIT Press, 2007.