Optimization methods for discriminative training

Jonathan Le Roux[1]*, Erik McDermott*

NTT Communication Science Laboratories, NTT Corporation
2-4 Hikaridai, Seika-cho, Soraku-gun, Kyoto-fu 619-0237, Japan.
leroux@hil.t.u-tokyo.ac.jp, mcd@cslab.kecl.ntt.co.jp

Abstract

Discriminative training applied to hidden Markov model (HMM) design can yield significant benefits in recognition accuracy and model compactness. However, compared to Maximum Likelihood based methods, discriminative training typically requires much more computation, as all competing candidates must be considered, not just the correct one. The choice of the algorithm used to optimize the discriminative criterion function is thus a key issue. We investigated several algorithms and used them for discriminative training based on the Minimum Classification Error (MCE) framework. In particular, we examined on-line, batch, and semi-batch Probabilistic Descent (PD), as well as Quickprop, Rprop and BFGS. We describe each algorithm and present comparative results on the TIMIT phone classification task and on the 230 hour Corpus of Spontaneous Japanese (CSJ) 30K word continuous speech recognition task.

1. Introduction

1.1. Motivation

Discriminative training has been applied to large scale speech recognition by several groups, using Maximum Mutual Information (MMI) [1], Minimum Phone Error (MPE) [2], or Minimum Classification Error (MCE) [3] as the theoretical framework for discriminative training. A central issue is the efficient optimization of the discriminative training criterion function.

In the context of MCE, the on-line, first-order, token-by-token Probabilistic Descent (PD) method [4] has been extremely popular. This approach is theoretically grounded, simple to implement, and shows excellent convergence speed. However, PD suffers from its on-line nature, making it hard to parallelize over multiple computers, and also from the fact that parameters such as the learning rate need to be tuned precisely. The use of growth transforms to achieve re-estimation algorithms similar in spirit to the Extended-Baum Welch algorithm for MMI has been proposed for MCE [5], but requires the use of a modified, "super-string" oriented MCE criterion function. See [6] for related work. Finally, the use of Quickprop [7], a heuristic second-order batch-oriented method, has yielded significant performance gains on some large-scale tasks [3]. Besides being rather heuristic, Quickprop also requires tuning of a gradient scaling parameter. Our aim was thus to find an algorithm which would surpass the existing ones with respect to theoretical justification, convergence speed, ease of parameter tuning, and final optimization effectiveness.

Here we will briefly review the MCE loss function, and then describe the optimization algorithms we applied to MCE training: PD-based methods [4, 8], modified versions of Quickprop [7], Rprop [9], and BFGS [10]. We will describe ideas for improving these methods, discuss the difficulties encountered, and give comparative speech recognition results.

[1] Jonathan Le Roux is currently at the Graduate School of Information Science and Technology, The University of Tokyo.

1.2. MCE misclassification measure and loss function

For a sequence of feature vectors, $\boldsymbol{x}_1^T = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$, the best Viterbi state sequence $\Theta^j = (\theta_1^j, \ldots, \theta_T^j)$ for string S_j is used to define the HMM discriminant function for S_j:

$$g_j(\boldsymbol{x}_1^T, \boldsymbol{\Lambda}) = \log P(S_j) + \sum_{t=1}^{T} \log a_{\theta_{t-1}^j \theta_t^j} + \sum_{t=1}^{T} \log b_{\theta_t^j}(\boldsymbol{x}_t), \tag{1}$$

where $P(S_j)$ denotes the prior (e.g. language model) probability of string S_j, a_{uv} a state transition probability, $b_s(\cdot)$ a Gaussian mixture used to model the observation probability of a feature vector in state s, and $\boldsymbol{\Lambda}$ the entire set of system parameters, consisting of the transition probabilities and the means, covariances and mixing weights used to define b_s. The misclassification measure for the correct string category S_k is defined as

$$d_k(\boldsymbol{x}_1^T, \boldsymbol{\Lambda}) = -g_k(\boldsymbol{x}_1^T, \boldsymbol{\Lambda}) + \log\Big(\frac{1}{M-1} \sum_{j \neq k}^{M} e^{g_j(\boldsymbol{x}_1^T, \boldsymbol{\Lambda})\psi}\Big)^{\frac{1}{\psi}}. \tag{2}$$

In the work described here, a large setting for ψ was used. The MCE loss function for a single token $\boldsymbol{x}_1^T$ is typically the composition of the misclassification measure with a sigmoid,

$$l\left(d_k(\boldsymbol{x}_1^T, \boldsymbol{\Lambda})\right) = \frac{1}{1 + e^{-\alpha d_k}}. \tag{3}$$

For a set of tokens, the MCE loss function is defined as the sum of the loss functions for each token.

2. On-line and Batch Probabilistic Descent

Probabilistic Descent (PD) [8] is a very simple and remarkably effective on-line optimization method. It consists in computing the gradient of the loss function for each token $\boldsymbol{x}$ and updating parameters in the opposite direction, by a proportion determined by a learning rate ϵ_t that decreases over time:

$$\boldsymbol{s} = -\epsilon_t \nabla_{\boldsymbol{\Lambda}} l(d_k(\boldsymbol{x}, \boldsymbol{\Lambda})). \tag{4}$$

The power of such on-line algorithms is that they exploit redundancies in the data, allowing learning to proceed very quickly. However, on-line algorithms cannot be parallelized using the straight-forward data-parallelism approach that applies to batch algorithms, where processors can accumulate gradient information for a given model over different subsets of the training data

before each model update. This means that although PD converges very quickly in terms of required number of iterations, in practice it suffers compared to parallelizable batch algorithms. On the other hand, the purely batch version of this algorithm may be slow to converge. We would like to be able to have a parallelizable version of on-line algorithms such as PD, without getting too far away from their on-line nature. One approach is to update the model every n tokens, allowing a measure of parallelization (subject to the possibility of increased between-process communication time overhead). We refer to this approach as "semi-batch". For all three approaches, on-line, semi-batch, and batch, the proper setting of the initial learning rate, ϵ_0, is important. A simple heuristic is to use the largest initial value that doesn't lead to unstable learning. In our experiments ϵ_t decreases linearly from ϵ_0 to 0.

3. Quickprop

3.1. Description

Quickprop [7] is a simple batch-oriented second-order optimization method. When the Hessian (or an approximation to it) is positive definite, we want to use it to compute the Newton step

$$\boldsymbol{s} = -\left(\Delta^2 F(\boldsymbol{\Lambda})\right)^{-1} \nabla F(\boldsymbol{\Lambda}), \tag{5}$$

which leads directly to the minimum of the quadratic approximation of a function F. If we are close to the minimum of F and the Hessian is positive definite, this step is likely to be effective.

In Quickprop, the Hessian is not analytically computed; only the diagonal part is used, and it is approximated by means of Eq. 6.

$$\frac{\partial^2 F(\boldsymbol{\Lambda}^{(p)})}{\partial \lambda_i^2} \approx \frac{\frac{\partial F(\boldsymbol{\Lambda}^{(p)})}{\partial \lambda_i} - \frac{\partial F(\boldsymbol{\Lambda}^{(p-1)})}{\partial \lambda_i}}{\Delta \lambda_i^{(p-1)}}, \tag{6}$$

where $\boldsymbol{\Lambda}^{(p)}$ denotes the parameter vector $\boldsymbol{\Lambda}$ at iteration p, λ_i the i-th component of $\boldsymbol{\Lambda}$ and $\Delta\lambda_i^{(p-1)}$ is the i-th component of the update step at iteration $p-1$.

Quickprop's principle is to look at the evolution of the sign of the gradient w.r.t. one parameter for successive iterations: if it is the same, we follow the gradient descent direction; if is different, a minimum is likely to exist in between the preceding and current values, and we are thus likely to be in a situation where the second-order approximation is good. We then want to use the approximation of the diagonal part of the Hessian. In order to have a somewhat safer update step, Quickprop uses the simple gradient multiplied by a "learning rate" ϵ as well, which makes it a compromise between Newton's method and simple gradient descent:

$$\boldsymbol{s} = -\left[\left(\Delta^2 F(\boldsymbol{\Lambda})\right)^{-1} + \epsilon\right] \nabla F(\boldsymbol{\Lambda}). \tag{7}$$

The proper setting of ϵ is important.

4. Rprop

4.1. Description

Rprop [9], which stands for "Resilient back-propagation", is a batch optimization algorithm well-known in the field of Artificial Neural Networks. Its basic principle is to eliminate the possibly harmful influence of the size of the partial derivative on the update step. As a remedy, only the sign of the derivative is considered to indicate the direction of the update $\Delta\lambda_i^{(p)}$ for the p-th epoch:

$$\Delta\lambda_i^{(p)} = \begin{cases} -\Delta_i^{(p)} & , \text{ if } \frac{\partial F(\boldsymbol{\Lambda}^{(p)})}{\partial \lambda_i} > 0 \\ +\Delta_i^{(p)} & , \text{ if } \frac{\partial F(\boldsymbol{\Lambda}^{(p)})}{\partial \lambda_i} < 0 \\ 0 & , \text{ else.} \end{cases} \tag{8}$$

The size $\Delta_i^{(p)}$ of the update is different for each parameter (though initialized uniformly at a user-determined $\Delta^{(0)}$), and evolves according to a very simple adaptation rule:

$$\Delta_i^{(p)} = \begin{cases} \eta^+ \cdot \Delta_i^{(p-1)} & , \text{ if } \frac{\partial F(\boldsymbol{\Lambda}^{(p-1)})}{\partial \lambda_i} \cdot \frac{\partial F(\boldsymbol{\Lambda}^{(p)})}{\partial \lambda_i} > 0 \\ \eta^- \cdot \Delta_i^{(p-1)} & , \text{ if } \frac{\partial F(\boldsymbol{\Lambda}^{(p-1)})}{\partial \lambda_i} \cdot \frac{\partial F(\boldsymbol{\Lambda}^{(p)})}{\partial \lambda_i} < 0 \\ \Delta_i^{(p-1)} & , \text{ else.} \end{cases} \tag{9}$$

where $0 < \eta^- < 1 < \eta^+$.

This means that each time the derivative w.r.t. a parameter changes sign, it is regarded as an indication that the last update was too large and that it jumped over a minimum. The update value is thus decreased by a factor η^-. If the derivative's sign does not change, the update value is increased in order to speed up the convergence in shallow regions.

We then have to choose the update parameters η^+ and η^-. A rather common choice is $\eta^+ = 0.5$ and $\eta^- = 1.2$.

It is possible for the standard version of the algorithm to skip over local minima. We thus implemented three versions of the Rprop algorithm, illustrated in Figure 1. The first one ("Standard Rprop") is the original version described by Equations 8 and 9. The second one is a modified version of "Rprop with weight-backtracking" described in [9]. It consists in not allowing update step adaptation in the epoch following an update step decrease. We also tried to decrease the step successively for two epochs after a change in sign of the derivative, enforcing a more thorough search. The "noise" of the change of all the other parameters actually makes it difficult to predict how the step should be forced to evolve after one or two updates; the "weight-backtracking" version turned out to be the most effective.

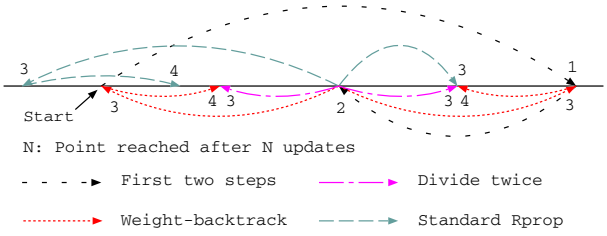

Figure 1: *Different versions of Rprop update*

4.2. Using a small or evolving update period

In addition to batch mode, we can also consider semi-batch versions, as well a hybrids of semi-batch and batch, e.g. an evolving update period which starts small but grows eventually to include the entire training set.

5. BFGS

5.1. Description of the update

The Newton techniques ideally require the computation of the full Hessian and its inversion. Both these tasks, as well as the

storage issue, make such techniques impossible to use directly in large scale problems. Quasi-Newton techniques, however, compute adaptively an approximation of the inverse of the Hessian. BFGS update [10] is one such method.

Using $\boldsymbol{g}_k = \nabla F(\boldsymbol{\Lambda}^{(k)})$, $\Delta\boldsymbol{\Lambda}^{(k)} = -\boldsymbol{H}_k\boldsymbol{g}_k$ the descent direction, $\boldsymbol{q}_k = \boldsymbol{g}_{k+1} - \boldsymbol{g}_k$, and $\boldsymbol{p}_k = \lambda_k\Delta\boldsymbol{\Lambda}^{(k)}$, where λ_k is a step factor that ensures $\boldsymbol{p}_k^T\boldsymbol{q}_k > 0$, the update can be written as follows:

$$\begin{aligned}\boldsymbol{H}_{k+1} &= \boldsymbol{H}_k - \frac{\boldsymbol{p}_k\boldsymbol{q}_k^T\boldsymbol{H}_k + \boldsymbol{H}_k\boldsymbol{q}_k\boldsymbol{p}_k^T}{\boldsymbol{p}_k^T\boldsymbol{q}_k} \\ &\quad + \left(1 + \frac{\boldsymbol{q}_k^T\boldsymbol{H}_k\boldsymbol{q}_k}{\boldsymbol{p}_k^T\boldsymbol{q}_k}\right)\frac{\boldsymbol{p}_k\boldsymbol{p}_k^T}{\boldsymbol{p}_k^T\boldsymbol{q}_k}. \qquad (10)\end{aligned}$$

It is then used as an approximation of the inverse of the Hessian in Equation (5).

5.2. Partial BFGS update

Storage and computation problems both make a direct use of this update impossible for large scale problems. Battiti [11] proposed to take the previous matrix to always be the identity, $\boldsymbol{H}_k = \boldsymbol{I}$. But due to this crude assumption, it might be impossible to get good results. The key idea to be able to apply BFGS update to MCE optimization was found in [12]. It is called "partial BFGS update", and consists in computing directly during s iterations the parameter update by means of a recurrence formula, without actually computing or storing the matrix $\boldsymbol{H}_k$, and then restarting the update. Keeping the last s vectors only is another possibility, but did not work well in our case and lacks theoretical justification. This approach is a different implementation of an idea described in [13]. If we set $\boldsymbol{r}_k = \boldsymbol{H}_k\boldsymbol{q}_k$, we have

$$\boldsymbol{H}_k\boldsymbol{g}_{k+1} = \boldsymbol{H}_k\boldsymbol{q}_k + \boldsymbol{H}_k\boldsymbol{g}_k = \boldsymbol{r}_k - \frac{\boldsymbol{p}_k}{\lambda_k}, \qquad (11)$$

$$\begin{aligned}\Delta\boldsymbol{\Lambda}^{(k+1)} &= -\boldsymbol{r}_k + \frac{\boldsymbol{p}_k}{\lambda_k} + \frac{\boldsymbol{p}_k\boldsymbol{r}_k^T\boldsymbol{g}_{k+1} + \boldsymbol{r}_k\boldsymbol{p}_k^T\boldsymbol{g}_{k+1}}{\boldsymbol{p}_k^T\boldsymbol{q}_k} \\ &\quad - \left(1 + \frac{\boldsymbol{q}_k^T\boldsymbol{r}_k}{\boldsymbol{p}_k^T\boldsymbol{q}_k}\right)\frac{\boldsymbol{p}_k\boldsymbol{p}_k^T\boldsymbol{g}_{k+1}}{\boldsymbol{p}_k^T\boldsymbol{q}_k}. \qquad (12)\end{aligned}$$

For $k = 1$, $\boldsymbol{r}_1 = \boldsymbol{g}_2 - \boldsymbol{g}_1$. As long as $k \leq s$, if we have stored the $\boldsymbol{p}_i$'s and $\boldsymbol{r}_i$'s, the sets of scalars $\alpha_i = \frac{1}{\boldsymbol{p}_i^T\boldsymbol{q}_i}$ and $\beta_i = \alpha_i(1 + \alpha_i\boldsymbol{q}_i^T\boldsymbol{r}_i)$ for $i < k$, then we can compute $\boldsymbol{r}_k$ using the following recurrence formula:

$$\boldsymbol{r}_k = \boldsymbol{q}_k + \sum_{i=1}^{k-1}\left(-\alpha_i\boldsymbol{p}_i\boldsymbol{r}_i^T\boldsymbol{q}_k - \alpha_i\boldsymbol{r}_i\boldsymbol{p}_i^T\boldsymbol{q}_k + \beta_i\boldsymbol{p}_i\boldsymbol{p}_i^T\boldsymbol{q}_k\right). \qquad (13)$$

5.3. Application to MCE: restarting the update or not

The "partiality" factor, controled by s, is very important when the number of updates is very large, but is not necessarily an issue in MCE batch optimization, as the number of epochs is typically less than a few dozen, and most often around ten. We can thus afford to have an exact BFGS update at low cost.

5.4. Line search

Once a search direction $\Delta\boldsymbol{\Lambda}^{(k)}$ has been determined by some algorithm, we might want to look for a $\lambda > 0$ which minimizes the function $F(\boldsymbol{\Lambda}^{(k)} + \lambda\Delta\boldsymbol{\Lambda}^{(k)})$, i.e. along this direction. While for Quickprop or Rprop, there is no particular need for a line search, in the case of BFGS, it is a part of the algorithm, but it does not need to be very accurate.

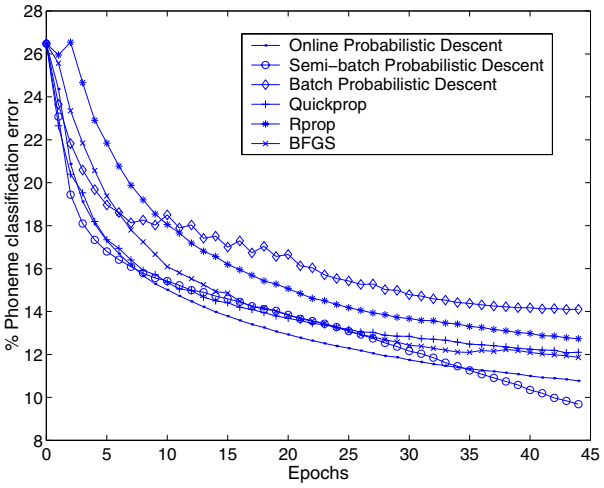

Figure 2: *Course of training for different optimization methods, TIMIT training set*

Performing a line search implies computing the value of the loss function, which in the MCE case is as costly as performing a full epoch. We thought it best to keep the updated step as long as it makes the loss decrease. If the loss increases, we restart from the previous step after dividing λ by 2.

The step given by the BFGS update ($\lambda_k = 1$) is actually very large, and needs to be reduced if we don't want to perform a line search at every step. We thus introduced a learning rate, which can be decreased after a chosen number of epochs. This learning rate and the speed of its decrease are the only parameters to tune in our version of BFGS.

6. Experiments

Two databases were used to evaluate the methods considered here: the TIMIT phone classification task [14], and the Corpus of Spontaneous Japanese (CSJ) LVCSR lecture speech transcription task [15].

6.1. TIMIT phone classification

The standard TIMIT training set (3696 utterances) and "core" test set (192 utterances) were used. The feature vectors used consist of 39 MFCCs, deltas and delta-deltas using a 25 ms window and a 10 ms shift rate. Though 48 phones are modeled, the common procedure of mapping these to 39 phones during testing was followed. Each phone was modeled with a 3 state, 8 Gaussian per state HMM.

The optimization methods described here were run using the same phone-level MCE loss function (Equ. 3), applied to each labeled speech segment. Each optimization method was run for 45 epochs (presentations of the training set).

For each method, training set phone classification error after each epoch is shown in Figure 2. Test set classification error rates are shown in Table 1 ("SB-PD" : Semi-batch PD; "B-PD": Batch PD; "Qprop": Quickprop). Two results for each method are listed: the result for the final model, after 45 epochs of optimization, and the best result over all iterations. The latter results correspond to the best we could do if we were to use a development set to select the model to use on the core test set. The best result obtained, 22.0% classification error, is a significant

Table 1: *TIMIT phone classification test set error rates*

	PD	SB-PD	B-PD	Qprop	Rprop	BFGS
E45	23.3	24.6	22.3	22.2	22.9	24.7
Best	22.4	23.5	22.2	22.0	22.7	23.0

improvement over the baseline Maximum Likelihood model's performance on the core test set, 29.2%.

6.2. Corpus of Spontaneous Japanese

The CSJ *A* set (male & female) consisting of about 186K utterances (approximately 230 hours of audio) was used for training. The trained models were tested on the standard test set of 10 lecture speeches, each from a different speaker, comprising 130 minutes of audio in total. The trigram language model WFST used in the recognition tests models 30,000 words and has a perplexity of 78.7. The feature vectors used consist of 38 MFCCs, deltas, and delta-deltas. A single acoustic model was used, with approximately 3000 states and 8 Gaussians per state, for a total of 24,000 Gaussian pdfs. MCE training was performed using a unigram WFST modeling about 48,000 words. This covers both all words in the training utterances and the 30K vocabulary used in the trigram language model used for testing. Beam search through the unigram WFST is fast enough that MCE training could be carried out without resorting to lattices. Starting with the baseline ML model, optimization based on semi-batch PD, Quickprop, semi-batch Rprop ("SB-Rprop", with an update period starting at 20,000 utterances, but doubling after every full presentation of the training set), and (batch) Rprop was carried out for 5-10 epochs. The model at the end of training was used for testing. The word error rate for each method on the training set using the unigram WFST, and the word error rate for each method on the test set using the standard trigram WFST, are shown in Table 2.

All the methods described here have parameters requiring tuning. For both TIMIT and CSJ, we found that the Rprop initial step was the easiest to tune: erring on the small side, a wide range of initial update steps yielded good convergence speed and final results.

7. Conclusion

Several optimization methods were applied to HMM design based on the MCE criterion function. For TIMIT, examining the evolution of performance on the training set, we see that online and semi-batch PD converge much more rapidly than batch PD. The differences between the other methods may or may not be significant. The differences on the test set may also not be significant, with the exception of semi-batch PD, which does not perform as well as the other methods. The classification error rate of 22.0% obtained for Quickprop is one of the best results for a standard HMM on this well-known task.

The results on the CSJ lecture speech transcription task show that for optimization on the training set, Rprop is very effective. Given the simplicity of the method and the ease of parameter tuning, Rprop seems like an attractive choice. However, Quickprop yielded the best test performance. More tests with these methods are needed to determine whether this pattern is robust on a task such as CSJ.

Altogether, the results obtained on both TIMIT and CSJ are good for all the methods evaluated, showing that the MCE criterion can be optimized effectively using very different methods.

Table 2: *Word error rates for the Corpus of Spontaneous Japanese.* Training set performance is based on a 48K word unigram; test set performance on a 30K word trigram.

	ML	SB-PD	Qprop	SB-Rprop	Rprop
Train	47.7	41.6	34.0	31.1	28.2
Test	23.8	22.7	22.0	22.9	22.7

8. References

[1] P.C. Woodland and D. Povey. Large scale discriminative training of hidden Markov models for speech recognition. *Computer Speech and Language*, 16:25–47, 2002.

[2] G. Evermann, H.Y. Chan, M.J.F. Gales, B. Jia, D. Mrva, P.C. Woodland, and K. Yu. Training LVCSR systems on thousands of hours of data. In *Proc. IEEE ICASSP*, volume 1, pages 209–212, 2005.

[3] E. McDermott and T. J. Hazen. Minimum Classification Error training of landmark models for real-time continuous speech recognition. In *Proc. IEEE ICASSP*, volume 1, pages 937–940, 2004.

[4] S. Katagiri, C-H. Lee, and B.-H. Juang. New discriminative training algorithms based on the generalized descent method. In *Proc. IEEE Worshop on Neural Networks for Signal Processing*, pages 299–308, 1991.

[5] X. He and W. Chou. Minimum Classification Error linear regression for acoustic model adaptation of continuous density HMMs. In *Proc. IEEE ICASSP*, volume 1, pages 556–559, 2003.

[6] Q. Li and B.-H. Juang. A new algorithm for fast discriminative training. In *Proc. ICASSP 2002*, volume 1, pages 97–100, May 2002.

[7] S. E. Fahlman. An Empirical Study of Learning Speed in Back-Propagation Networks. Technical report, Canergie Mellon University, 1988.

[8] S.-I. Amari. A theory of adaptive pattern classifiers. *IEEE Transactions on Electronic Computers*, EC-16:299–307, June 1967.

[9] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proc. of the IEEE Intl. Conf. on Neural Networks*, pages 586–591, San Francisco, CA, 1993.

[10] C. G. Broyden, J. E. Dennis, Jr., and Jorge J. More. On the local and superlinear convergence of quasi-Newton methods. *Journal of the Institute of Mathematics and its Applications*, 12:223–245, 1973.

[11] R. Battiti. First- and Second- Order Methods for Learning: Between Steepest Descent and Newton's Method. *Neural Computation*, 4:141–166, 1992.

[12] Kazumi Saito and Ryohei Nakano. Partial BFGS update and efficient step-length calculation for three-layer neural networks. *Neural Computation*, 9(1):123–141, 1997.

[13] Jorge Nocedal. Updating quasi-Newton matrices with limited storage. *Mathematics of Computation*, 35(151):773–782, July 1980.

[14] A.J. Robinson. Application of Recurrent Nets to Phone Probability Estimation. *IEEE Transactions on Neural Networks*, 5(2):298–305, March 1994.

[15] T. Kawahara, H. Nanjo, T. Shinozaki, and S. Furui. Benchmark test for speech recognition using the corpus of spontaneous japanese. In *Proc. of the Spontaneous Speech Processing & Recognition Workshop*, pages 135–138, Tokyo, 2003.