Parallel training of Deep Neural Networks with Natural Gradient and Parameter Averaging

Daniel Povey[*] **Xiaohui Zhang** **Sanjeev Khudanpur**[*]
Center for Language and Speech Processing, Johns Hopkins University
and (*) Human Language Technology Center of Excellence, Johns Hopkins University

Abstract

We describe the neural-network training framework used in the Kaldi speech recognition toolkit, which is geared towards training DNNs with large amounts of training data using multiple GPU-equipped or multi-core machines. In order to be as hardware-agnostic as possible, we needed a way to use multiple machines without generating excessive network traffic. Our method is to average the neural network parameters periodically (typically every minute or two), and redistribute the averaged parameters to the machines for further training. Each machine sees different data. By itself, this method does not work very well. However, we have another method, an approximate and efficient implementation of Natural Gradient for Stochastic Gradient Descent (NG-SGD), which seems to allow our periodic-averaging method to work well, as well as substantially improving the convergence of SGD on a single machine.

1 Introduction

In this paper we introduce two ideas: a data-parallel training method based on periodically averaging the parameters of separate SGD runs; and an efficient and practical implementation of Natural Gradient for Stochastic Gradient Descent (NG-SGD) for deep neural networks.

In Section 2 we give some background on our problem setting, which is Deep Neural Networks (DNNs) applied to speech recognition– although our ideas are more general than this. In Section 3 we introduce the parallel training method. In Section 4 we describe the general ideas behind our natural gradient method, although most of the technical details have been relegated to appendices in the Supplementary Material. In this paper we don't give any proofs, but we do discuss in Section 5 what we think we can and can't be proven about our methods. Section 6 has experiments, in which we plot the convergence of SGD with and without natural gradient and parallelism. We conclude in Section 7.

There are two versions of our NG-SGD method: a "simple" version and an "online" one. Technical details for these are in Appendices A and B respectively. Appendix C has more background information on our DNN implementation.

2 Problem setting

When training DNNs for speech recognition, the immediate problem is that of classifying vectors $\mathbf{x} \in \mathbb{R}^D$ as corresponding to discrete labels $y \in \mathcal{Y}$. The dimension D is typically several hundred, with $\mathbf{x}$ being derived from short-time spectral properties of the acoustic signal; and $\mathcal{Y}$ corresponds to clustered HMM states of context-dependent phones, $|\mathcal{Y}| \simeq 5000$ being typical. Each $(\mathbf{x}, y)$ pair corresponds to a single *frame* of speech data; frames are typically extracted at the rate of 100 per second, with a duration of 25 milliseconds, and $\mathbf{x}$ contains spectral information from several adjacent frames spliced together [1]. We are ultimately not just interested in the top y on each frame, but in the log-probabilities $\log p(y|\mathbf{x})$ for all y, since we will use them as costs in a Viterbi search for a path corresponding to the most likely word sequence. The objective function for training is the sum, over all frames of training data, of the log-probability of y given $\mathbf{x}$: $\sum_i \log p(y_i|\mathbf{x}_i)$. The supervision labels y are derived from a Viterbi alignment of a Hidden Markov Model (HMM) derived from the reference word sequence of each training utterance.

Preliminary work. Under review by AISTATS 2014. Do not distribute.

3 SGD with parameter averaging

3.1 Parameter-averaging overview

The parameter-averaging aspect of our training is quite simple. We have N machines (e.g. $N = 4$) each doing SGD separately with different randomized subsets of the training data, and we allow their parameters to gradually diverge. After each machine has processed a fixed number of samples K (typically $K = 400\,000$), we average the parameters across all the jobs and re-distribute the result to the machines. (In practice we do this by spawning new jobs in GridEngine or in whatever job management system we are using). This is repeated until we have processed all the data for a specified number of epochs, e.g. 10.

We define an *outer iteration* of training as the time it takes for each job to process K training examples. There are one or more outer iterations per epoch, depending on the quantity of training data.

We find it useful to define the effective learning rate of the parallel SGD procedure as the learning rate η_t being used by the individual jobs, divided by the number of jobs N. As we increase the number of jobs N, in order to get a linear speed up we need to increase the learning rate proportional to N so that the effective learning rate stays the same. We have not done any very formal analysis on this, but the basic idea is that when we do the parameter averaging, the parameter update from any individual SGD job gets diluted N-fold.

3.2 CPU versus GPU-based SGD

Each machine in our parallel computation implements SGD. We have two versions of this, one for GPU and one for CPU. The GPU-based computation is standard minibatch-based SGD, typically with 512 examples per minibatch.

In the CPU-based computation, each job uses a specified number of threads (typically 16) in order to take advantage of multi-core processors. Similar to Hogwild! [2], we do no parameter locking. In order to prevent divergence, each thread processes relatively small minibatches - typically, of size 128.

We should mention at this point that in our formulation, we sum the gradients over the elements of the minibatch, rather than averaging: this ensures that we make the same amount of progress per sample, regardless of minibatch size, and so gives more consistent results when changing the minibatch size. The need to limit the minibatch size in the multithreaded case can be understood as follows: think of the effective minibatch size as being the minibatch size times the number of threads. The product of the learning rate η with the effective minibatch size is relevant for stability of the SGD update: if it becomes too large, there is increased danger of divergence.

We typically use the GPU-based method, because in our experience a GPU can process data many times faster than even a 16-threaded process running on CPUs. However, aside from speed, the two methods give very similar results.

3.3 Data organization and sequential data access

On spinning hard disks, sequential data access can be orders of magnitude more efficient than random data access or access to small files. In the Kaldi toolkit [3], we try very hard to ensure that any high-volume data access takes the form of sequential reads or writes on large files.

For neural network training, we keep data access sequential by dumping pre-randomized "training examples" to disk. Each training example corresponds to a class label together with the corresponding input features, including left and right temporal context as needed by the network. The randomization is done just once for the entire data, and the data is accessed in the same order on each epoch. This is probably not ideal from the point of view of the convergence of SGD, but our expectation is that for large amounts of data the same-order access will not affect the results noticeably.

We break up the training data into N by M rougly equal-sized blocks, where N is the number of parallel jobs, specified by the user (typically $4 \leq N \leq 8$), and $M \geq 1$ is the number of "outer iterations per epoch", which is chosen to ensure that the number of samples processed per iteration is close to a user-specified value K (e.g. $K = 400\,000$). The process of randomly distributing the data into N by M blocks, and ensuring that the order is randomized within each block, is done in parallel; we won't give further details here, because the problem is straightforward and there is nothing particularly special about our method.

3.4 Other aspects of our SGD implementation

At this point we provide some more details of other relevant features of our SGD implementation, namely the learning rate schedule and the way we enforce a maximum parameter change to prevent divergence.

There are some other, less directly relevant issues which we have relegated to Appendix C: namely, generalized model averaging (C.2), mixture com-

ponents a.k.a. sub-classes (C.3), input data normalization (C.4), parameter initialization (C.5), sequence training (C.6), and online decoding with iVectors (C.7).

3.4.1 Learning rate schedule

It was found in [4] that when training DNNs for speech recognition, an exponentially decreasing learning rate works well, and we independently found the same thing. We generally use a learning rate that decreases by a factor of 10 during training, on an exponential schedule. Unless mentioned otherwise, for experiments reported here the learning rate starts at 0.01 and ends at 0.0001. We specify the number of epochs in advance; it is typically a number in the range 4 to 20 (if we have more data, we train for fewer epochs).

3.4.2 Maximum parameter change

A common pathology when doing SGD for deep learning is that during training, the parameters will suddenly start getting very large and the objective function will go to negative infinity. This is known as parameter divergence. The normal solution is to decrease the learning rate and start the training again, but this is a very inconvenient. To avoid this pathology, we modified the SGD procedure to enforce a maximum parameter change per minibatch. This limit tends to be active only early in training, particularly for layers closer to the output. We have provided further details on this in Appendix C.1.

4 Natural gradient for SGD

In this section we describe our natural-gradient modification to SGD, in which we scale the gradients by a symmetric positive definite matrix that is an approximation to the inverse of the Fisher matrix.

Technically speaking, Natural Gradient means taking a step along the gradient of a Riemannian parameter surface, which follows a curving path in conventional parameter space and which is extremely hard to compute. However, previous work [5, 6] has used the term "Natural Gradient" to describe methods like ours which use an an approximated inverse-Fisher matrix as the learning rate matrix, so we follow their precedent in calling our method "Natural Gradient".

4.1 We can replace the scalar learning rate in SGD with a matrix

In SGD, the learning rate is often assumed to be a scalar η_t, decreasing with time, and the update equation is something like

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta_t \mathbf{g}_t$$

where $\mathbf{g}_t$ is the objective-function gradient sampled on time t (e.g. computed from a training sample or a minibatch). However, it is possible to replace this scalar with a symmetric positive definite matrix, and we can write instead:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta_t \mathbf{E}_t \mathbf{g}_t \tag{1}$$

with $\mathbf{E}_t$ the matrix component of learning-rate; it is more convenient for proofs to keep η_t separate rather than absorbing it into $\mathbf{E}_t$. It acceptable for $\mathbf{E}_t$ to be random: if we can bound the eigenvalues of $\mathbf{E}_t$ above and below, by positive constants known in advance, and $\mathbf{E}_t$ and $\mathbf{g}_t$ are independently sampled given the parameter $\boldsymbol{\theta}$, then we can prove convergence under the same kinds of conditions as if we were using a scalar learning rate [7, Sec. 4.2.2]

In general, the learning-rate matrix should not be a function of the data sample which we are currently processing, or it may prevent convergence to a local optimum. As an example of this, a matrix that was systematically smaller for a particular type of training data would clearly bias the learning by downweighting that data.

4.2 The inverse Fisher matrix is a suitable learning-rate matrix

There are reasons from statistical learning theory, related to the Natural Gradient idea [8], why we may want to set the learning-rate matrix $\mathbf{E}_t$ to the inverse of the Fisher information matrix. See for example, [9] and [6]. The Fisher matrix is most directly defined for situations where we are learning a distribution, as opposed to classification problems such as the current one. Suppose x, which may be discrete or continuous, is the variable whose distribution we are modeling, and $f(x; \boldsymbol{\theta})$ is the probability or likelihood of x given parameters $\boldsymbol{\theta}$, then the Fisher information matrix $\mathcal{I}(\boldsymbol{\theta})$ is defined as the expected outer product (second moment) of the derivative of the log-probability w.r.t. the parameters, i.e. of

$$\frac{\partial}{\partial \boldsymbol{\theta}} \log f(x; \boldsymbol{\theta}).$$

This derivative is called the "score" in information theory. Part of the justification for this use of the Fisher matrix is that, under certain conditions, the Fisher matrix is identical to the Hessian; and it is obvious why the inverse Hessian would be a good gradient descent direction. These conditions are quite stringent, and include that the model is correct and $\boldsymbol{\theta}$ is at the

value corresponding to the true data distribution; but even if these conditions do not apply, the Fisher information matrix is in some sense "dimensionally" the same as the Hessian– that is, it transforms the same way under changes of parameterization– so its inverse may still be a good choice of learning-rate matrix.

It is quite easy to generalize the notion of the Fisher matrix to a prediction task $p(y; x, \boldsymbol{\theta})$. We write $p(y, x; \boldsymbol{\theta}) = q(x)p(y; x, \boldsymbol{\theta})$ for a data distribution $q(x)$ that we assume to be independently known (and not a function of $\boldsymbol{\theta}$). It is not hard to see that the score equals just $\frac{\partial}{\partial \boldsymbol{\theta}} \log f(x; y, \boldsymbol{\theta})$; since $q(x)$ does not depend on $\boldsymbol{\theta}$, there is no additional term involving $q(x)$. The expectation that we take when computing the Fisher matrix is taken over the joint distribution of x and y. This argument also appears in [6, Section 3].

Still more generally, we can compute a quantity that is analogous to Fisher matrix for any objective function, even one that does not represent a log-probability or log-likelihood; and we will still have a matrix that transforms in the same way as the Hessian under changes of variables - i.e. its inverse may still be a reasonable choice for a learning-rate matrix.

4.3 We need to approximate the Fisher matrix in practice

For large-scale problems, such as DNNs for speech recognition with millions of parameters, even one inversion of the Fisher matrix is impractical because it would take time $O(n^3)$ in the parameter dimension. However, it may be practical to deal with factored forms of it. There has been previous literature on this. In [6], the Fisher matrix was divided into diagonal blocks and each block was approximated by a low-rank matrix. The idea of diagonal blocks was also explored in [10], with one block per weight matrix; our approach uses the same idea. In the unpublished manuscript [11] (some of the material in which was later published in [5]), the authors attempted to show analytically that under certain quite strong assumptions, the Fisher matrix for a single-hidden-layer neural network has the form of a Kronecker product. Although we are skeptical of this, and anyway we are interested in more general networks than they considered, the Kronecker product does also appear in our factorization of the Fisher matrix.

We should note that there are ways to use Natural Gradient without factorizing the Fisher information matrix, if one is willing to accept a significantly increased time per iteration. See for example [12], which uses a truncated Newton method to approximate multiplication by the inverse of the Fisher matrix.

4.4 Our factorization of the Fisher matrix

Our factored form of the Fisher matrix is as follows: given a neural network with I weight matrices, we divide the Fisher matrix into I diagonal blocks, one for each weight matrix. Consider the i'th diagonal block of the Fisher matrix, corresponding to the parameters of a weight matrix $\mathbf{W}_i$, and assume that there is no separate bias term (we can append a 1 to the inputs and include the bias term in the weight matrix). The i'th block of the Fisher matrix is a Kronecker product of two symmetric positive definite matrices: $\mathbf{A}_i$, whose dimension is the input (row) dimension of $\mathbf{W}_i$, and $\mathbf{B}_i$, whose dimension is the output (column) dimension of $\mathbf{W}_i$. We further factorize the matrices $\mathbf{A}_i$ and $\mathbf{B}_i$ as a low-rank symmetric matrix plus a multiple of the identity matrix. We write the approximated Fisher matrix in the form

$$\mathbf{F} = \mathrm{diag}\left(\mathbf{A}_1 \otimes \mathbf{B}_1, \mathbf{A}_2 \otimes \mathbf{B}_2, \ldots, \mathbf{A}_I \otimes \mathbf{B}_I\right) \qquad (2)$$

where $\mathbf{A}_i$ and $\mathbf{B}_i$ are each factorized in the form $\lambda\mathbf{I} + \mathbf{X}\mathbf{X}^T$. The order in which $\mathbf{A}_i$ and $\mathbf{B}_i$ appear in the Kronecker product depends on the way in which we vectorize the weight matrices– row-wise or column-wise. In practice we don't ever deal explicitly with these Kronecker products or vectorized weight matrices in the algorithm, so this choice doesn't matter. It is not hard to show that if the Fisher matrix can be factored this way, then its inverse can be factored the same way.

4.5 How we estimate the Fisher matrix

We have two different methods for estimating the factorized Fisher matrix:

- Simple method: We estimate the Fisher matrix from the *other* samples in the minibatch we are currently training on, holding out the current sample to avoid bias. This can be done surprisingly efficiently. Details are in Appendix A.

- Online method: We estimate the Fisher matrix from all previous minibatches, using a forgetting factor to downweight minibatches that are distant in time. Details are in Appendix B.

We generally use the online method as it is significantly faster on GPUs and usually seems to lead to faster learning, probably due to the less noisy estimate of the Fisher matrix. We describe the simple method because it is easier to understand and helps to motivate the online method.

4.6 Operation on vectors

Although we describe our Fisher matrix as as a Kronecker product, we do not have to explicitly construct this product in our code.

Suppose that we process the training examples one at a time. The SGD update for the i'th weight matrix is:

$$\mathbf{W}_{ti} = \mathbf{W}_{t-1,i} + \eta_t \mathbf{x}_{ti} \mathbf{y}_{ti}^T$$

where $\mathbf{x}_{ti}$ is the input to the i'th weight matrix computed at the current sample, and $\mathbf{y}_{ti}$ is the derivative of the objective function with respect to the output of the i'th weight matrix. These quantities naturally occur in backpropagation.

In our natural gradient method, this is modified as follows:

$$\mathbf{W}_{ti} = \mathbf{W}_{t-1,i} + \eta_t \mathbf{A}_{ti}^{-1} \mathbf{x}_{ti} \mathbf{y}_{ti}^T \mathbf{B}_{ti}^{-1},$$

where $\mathbf{A}_{ti}$ and $\mathbf{B}_{ti}$ are factors of the Fisher matrix. It is easy to show that this is equivalent to multiplying the parameter step by the inverse of the Fisher matrix formed from the $\mathbf{A}$ and $\mathbf{B}$ quantities as in Equation (2).

4.7 Operation on minibatches

Rather than processing training examples one at a time, we process them in minibatches (e.g. 512 at a time). Instead of vector-valued derivatives $\mathbf{x}_{ti}$ and $\mathbf{y}_{ti}$, we now have matrices $\mathbf{X}_{ti}$ and $\mathbf{Y}_{ti}$, each row of which corresponds to one of the $\mathbf{x}$ or $\mathbf{y}$ quantities (t is now the index for the minibatch). The update is now as follows:

$$\mathbf{W}_{ti} = \mathbf{W}_{t-1,i} + \eta_t \mathbf{X}_{ti}^T \mathbf{Y}_{ti} \tag{3}$$

and note that unlike some authors, we don't divide the gradients by the minibatch size– this makes it easier to tune the minibatch size and learning rate independently. The update now has the form

$$\mathbf{W}_{ti} = \mathbf{W}_{t-1,i} + \eta_t \bar{\mathbf{X}}_{ti}^T \bar{\mathbf{Y}}_{ti}, \tag{4}$$

with the bar indicating modified $\mathbf{X}$ and $\mathbf{Y}$ quantities. In the online version of our natural gradient method, we can write these as:

$$\bar{\mathbf{X}}_{ti} = \mathbf{X}_{ti} \mathbf{A}_{ti}^{-1} \tag{5}$$

$$\bar{\mathbf{Y}}_{ti} = \mathbf{Y}_{ti} \mathbf{B}_{ti}^{-1}, \tag{6}$$

but in the simple method, because the $\mathbf{A}$ and $\mathbf{B}$ matrices are estimated from the *other* elements in the minibatch, we can't write it this way– it is a separate matrix multiplication for each row of $\mathbf{X}$ and $\mathbf{Y}$– but it can still be computed efficiently; see Appendix A.

In programming terms, we can describe the interface of the core natural-gradient code as follows:

- Simple method: Given a minibatch of vectors $\mathbf{X}_{ti}$ with each row being one element of the minibatch, estimate the Fisher-matrix factors by holding out each sample, do the multiplication by their inverses, and return the modified vectors $\bar{\mathbf{X}}_{ti}$.

- Online method: Given a minibatch of vectors $\mathbf{X}_{ti}$ and a previous Fisher-matrix factor $\mathbf{A}_{t-1,i}$, compute $\bar{\mathbf{X}}_{ti} = \mathbf{X}_{ti} \mathbf{A}_{t-1,i}^{-1}$ and the updated Fisher-matrix factor $\mathbf{A}_{ti}$.

The interface of the natural gradient code works the same with the $\mathbf{Y}$ and $\mathbf{B}$ quantities, as with $\mathbf{X}$ and $\mathbf{A}$. We call the interface above $2I$ times for each minibatch: twice for each weight matrix in the network.

4.8 Scaling the factors

In both natural gradient methods, we want to prevent the Fisher-matrix multiplication from affecting the overall magnitude of the update very much, compared with the step-sizes in standard SGD. There are several reasons for this:

- Early in training, the $\mathbf{x}$ and $\mathbf{y}$ quantities may be very small or zero, leading to huge or infinite inverse-Fisher matrices.

- The conventional convergence-proof techniques require that the matrix component of the learning rate matrix should have eigenvalues bounded above and below by constants known in advance, which we cannot guarantee if we use an unmodified Fisher matrix.

- Empirically, we have found that it is hard to prevent parameter divergence if we use the real, unscaled Fisher matrix.

Our method is to scale the $\bar{\mathbf{X}}_{ti}$ and $\bar{\mathbf{Y}}_{ti}$ quantities so that they have the same Frobenius norm as the corresponding inputs $\mathbf{X}_{ti}$ and $\mathbf{Y}_{ti}$. We will introduce notation for this in the Appendices.

This scaling introduces a slight problem for convergence proofs. The issue is that each sample can now affect the value of its own learning-rate matrix (via the scalar factor that we use to rescale the matrices). As we mentioned before, it is not permissible in general to use a per-sample learning rate that is a function of the sample itself. However, we don't view this as a practical problem because we never use a minibatch size less than 100, so the resulting bias is tiny.

4.9 Smoothing the Fisher matrix factors with the identity

In both versions of NG-SGD, we smooth our estimates of the factors of the Fisher matrix by adding a multiple of the identity matrix before inverting them. In the simple method this is necessary because in general the Fisher matrix estimated from the minibatch will not be full rank. In the online method it is not strictly necessary because we deal with a factorization of the Fisher matrix that already contains a multiple of the unit matrix, but we found that by adding an additional multiple of the unit matrix, as for the simple method, we can improve the convergence of the SGD training. In both cases the smoothing is of the following form. If $\mathbf{S} \in \mathbb{R}^{D \times D}$ is a Fisher matrix factor estimated directly from data as the uncentered covariance of the $\mathbf{x}$ or $\mathbf{y}$ quantities, then instead of using $\mathbf{S}$ as the Fisher-matrix factor $\mathbf{A}$ or $\mathbf{B}$, we use instead $\mathbf{S} + \beta\mathbf{I}$, where

$$\beta = \frac{\alpha}{D} \max(\mathrm{tr}\,(\mathbf{S}), \epsilon) \tag{7}$$

where $\epsilon = 10^{-20}$ is used to stop the smoothed $\mathbf{S}$ from ever being exactly zero. That is, we smooth the Fisher with the identity matrix scaled by α times the average diagonal element of $\mathbf{S}$. We found in tuning experiments that the relatively large value $\alpha = 4$ is suitable under a wide range of circumstances, for both the simple and online methods, and even for settings where the noise in $\mathbf{S}$ should not be a big problem– e.g. for large minibatch sizes. Our interpretation is that when α is fairly large, we are using a smaller than normal learning rate only in a few directions where the $\mathbf{x}$ or $\mathbf{y}$ quantities have quite high covariance, and a relatively constant learning rate in all the remaining directions.

5 What we think we can, and can't, prove

Although this is not a theoretical paper, we would like to say what we think is, and is not, possible to prove about our methods.

5.1 Our factorization of the Fisher matrix

If we assume that the distribution of the $\mathbf{x}$ and $\mathbf{y}$ quantities is Gaussian and independent (between $\mathbf{x}$ and $\mathbf{y}$ for a single layer, and between layers), then it should not be hard to show that the Fisher matrix has the form of (2), where the $\mathbf{A}_i$ and $\mathbf{B}_i$ quantities correspond to the uncentered covariances of the $\mathbf{x}$ and $\mathbf{y}$ quantities, and that the inverse-Fisher has the same form, with the $\mathbf{A}_i^{-1}$ replacing $\mathbf{A}_i$ and $\mathbf{B}_i^{-1}$ replacing $\mathbf{B}_i$.

We don't believe that the Fisher matrix would have this exact form in practice, but we do believe that it's a reasonable factorization. One could try to show this experimentally as follows, for a small task. One could make a linear change of variables to make our approximated Fisher matrix equal the unit matrix, and then try to measure the eigenvalue distribution of the full Fisher matrix in the new co-ordinates. We believe that the eigenvalue distribution of the transformed Fisher matrix would probably be much more closerly centered around 1 than before the change of variables. Our motivation for this work is a practical one, so we have not allocated effort towards this type of experiment.

5.2 The convergence of our preconditioned SGD procedure

Regarding the convergence of SGD using our factored-Fisher learning rate matrices, the most we think is provable is that a slightly modified form of this method would converge under similar conditions to unmodified SGD. The smoothing with constant $\alpha > 0$ can give us a bound on the ratio of the largest to smallest eigenvalues of the $\mathbf{A}$ and $\mathbf{B}$ factors; using this together with the rescaling of Section 4.8, we can bound from above and below the eigenvalues of the rescaled $\mathbf{A}$ and $\mathbf{B}$ factors. By multiplying these together, we can get lower and upper bounds on the eigenvalues of the overall inverse-Fisher matrix that we use as the learning-rate matrix $\mathbf{E}_t$.

It is necessary for the Fisher matrix to be randomly chosen independent of the identity of the current sample. Unfortunately this is not quite true due to the rescaling being done at the minibatch level; we mentioned in Section 4.8 that this would be a problem for proofs. As mentioned, it would be easy to use the rescaling factor from the previous minibatch; this gives us back the independence, but at the cost of no longer having such easy bounds on the upper and lower eigenvalues of the rescaled $\mathbf{A}$ and $\mathbf{B}$ factors. Alternately, one could keep the algorithm as it is and try to prove instead that the parameter value we converge to will not differ very much in some sense from an optimum of the true objective function, as the minibatch size gets large.

5.3 Online update of a low-rank covariance matrix

There might be some interesting things to say about our online natural gradient method, described in Appendix B, in which estimate the uncentered covariance matrices $\mathbf{A}$ and $\mathbf{B}$ in a factored form as $\lambda\mathbf{I} + \mathbf{X}\mathbf{X}^T$. Our online estimation of the covariance matrices involves multiplying $\mathbf{X}$ by a weighted combination of (a) the observed covariance matrix from the current minibatch, and (b) the previous value of our factored

approximation to it; it is like a matrix version of the power method [13].

Probably the analysis would have to be done initially in the steady state (i.e. assuming the parameter vector $\boldsymbol{\theta}$ is constant). If in addition we assume infinite minibatch size so that the covariance matrix equals its expected value, we are confident that we could show that the only stable fixed point of our update equations gives us in some suitable sense the closest approximation to the covariance; and, with a little more effort, that our updates will converge with probability 1 to that best approximation.

The analysis for finite minibatch size would have to involve different methods. Because of the noise and the finite forgetting factor, we would never converge to the true value; but it might be possible to define some objective function that measures some kind of goodness of approximation, and then say something about the convergence of the distribution of that objective function. Our update method is invariant to orthogonal transformations of the vectors whose covariance we are approximating, so we can easily reduce to the diagonal-covariance case to make analysis easier.

6 Experiments

We show experiments on a speech recognition setup called Fisher English [1], which is English-language conversational telephone speech, sampled at 8kHz, and transcribed in a quick but relatively low-quality way. The total amount of training data is 1600 hours (only including transcribed segments, i.e. not the silent other half of the telephone conversation). We test on a held-out subset of the data, about 3.3 hours long, that we defined ourselves.

6.1 System details and Word Error Rate performance

Table 1: Word Error Rates (Fisher dev set)

Model	%WER
GMM	31.07
DNN1	23.66
DNN2	23.79

Our main results are convergence plots, but to give the reader some idea of the ultimate results in Word Error Rate, we show some results in Table 1. The Word Error Rates may seem on the high side, but this is mainly due to the difficulty of the data and the quick transcription method used on this data.

The GMM system is based on MFCC features, spliced across ± 3 frames and processed with LDA+MLLT to 40-dimensional features, then adapted with feature-space MLLR (fMLLR) in both training and test time. See [3] for an explanation of these terms and the normal system build steps. All these systems used the same phonetic context decision tree with 7 880 context-dependent states; the GMM system had 300 000 Gaussians in total.

The DNN1 system is built and tested on top of the speaker-adapted features, so it requires a first pass of decoding and adaptation with the GMM system. The 40-dimensional features from GMM1 are spliced across ± 4 frames of context and used as input to the DNN. DNN1 is a p-norm DNN [14] with 5 hidden layers and p-norm (input, output) dimensions of (5000, 500) respectively, i.e. the nonlinearity reduces the dimension tenfold. We use 15 000 "sub-classes" (see Section C.3 for explanation), and the number of parameters is 19.3 million. It is trained for 12 epochs with learning rate varying from 0.08 to 0.008, trained with 8 parallel jobs with online natural gradient SGD (NG-SGD). For both this and the DNN2 system, we trained with $K = 400\,000$ samples per outer iteration for each machine.

The DNN2 system is trained for our online decoding setup (see Appendix C.7), which is geared towards applications where reduced latency is important and audio data must be processed strictly in the order it is received. The input features are equivalent to unadapted, un-normalized 40-dimensional log-mel filterbank features, spliced for ± 7 frames, plus a 100-dimensional iVector representing speaker characteristics, extracted from only the speaker's audio up to and including the current time. For the results shown here, we include previous utterances of the same speaker in the same conversation when computing the iVector. Because this system is intended for real-time decoding on a single CPU, we limit the number of parameters by using only 4 hidden layers, p-norm (input, output) dimensions of (350, 3500), and 12 000 sub-classes, for a total of 10.4 million parameters. It was trained using online NG-SGD with 6 parallel jobs for 5 epochs, with the learning rate decreasing exponentially from 0.01 to 0.001. All our experiments below are based on this setup.

Our server hardware is fairly typical: the majority of them are Dell PowerEdge R720 servers with two Intel Xeon E5-2680v2 CPUs having with 10 cores each, running at 2.8GHz; and with a single NVidia Tesla K10 GPU card, providing two GPUs– each GPU cor-

[1] Linguistic Data Consortium (LDC) catalog numbers LDC2004S13, LDC2005S13, LDC2004T19 and LDC2005T19

responds to a single machine in our notation, and it becomes incidental that they are co-located. We also have some similar machines with K20 GPU cards, and when reporting time taken, we report the slightly more optimistic figures obtained from running the same jobs on the faster K20 GPUs.

6.2 Convergence plots

Our main result is in Figure 1 (best viewed in color), where we plot the objective function versus amount of training data processed, for our parallel training method with and without natural gradient, and with 1, 2, 4, 8 and 16 jobs. In order to keep the effective learning rate (Section 3.1) constant, we make the initial/final learning rates proportional to the number of jobs, with the default learning rates of 0.01 to 0.001 corresponding to the 6-job case. We only show the first 3 epochs of our 5-epoch training procedure; lines that end earlier are unfinished experiments.

Our natural gradient method always helps– the NG-SGD curves are all above the plain-SGD curves. Also, when using online natural-gradient, all the curves shown in Figure 1 are close to each other, i.e. after processing the same amount of data with different numbers of jobs we get about the same objective function; however, the 8- and 16-job runs converge a little slower. Thus, for small N we are getting a linear speed up in the number N of machines, because the time taken per epoch is proportional to $1/N$. As N gets larger than around 4 we need more epochs to get the same improvement, so the speedup becomes sublinear. The plot also shows that the simple and online natural gradient converge about the same (only tested with one job).

Figure 2 shows these same plots with time as the x-axis. This is a simulated clock time, obtained by multiplying the time taken for each "outer iteration" of training, by the number of outer iterations; the actual clock time depends on queue load. The time per outer iteration was 88 seconds for plain SGD, 93 seconds for online NG-SGD, and 208 seconds for plain NG-SGD, all measured on a K20 GPU. The circles mark the end of training, after 5 epochs (some experiments were not run to completion).

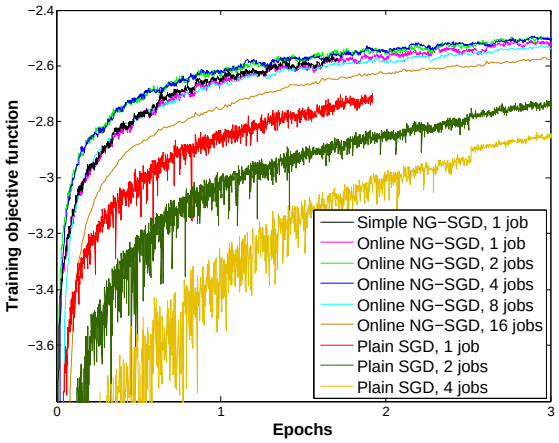

Figure 1: Objective function vs. epochs

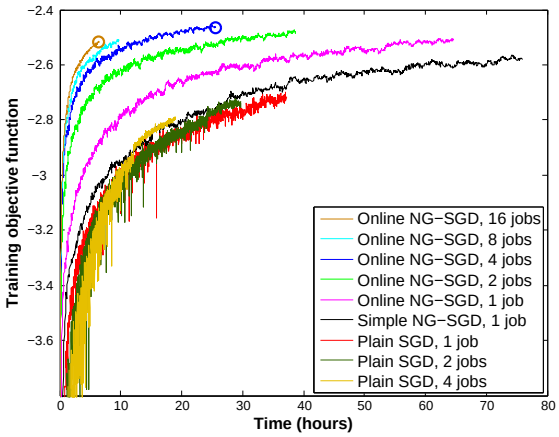

Figure 2: Objective function vs. time

7 Conclusions

We have introduced an efficient Natural Gradient version of SGD training (NG-SGD). We have shown experimentally that not only does the method improve the convergence versus plain SGD, it also makes it possible for us to to use very simple parallelization method where we periodically average parameters from multiple SGD runs. Although we only show results from one setup, we are confident based on past experience that it holds true for other types of neural network and improves our final results (Word Error Rate) as well as convergence speed.

We do not have a very good explanation why our parallel training method only works when using Natural Gradient, except to say that the statements in [12] that NG prevents large parameter steps and is more robust to reorderings of the training set, may be relevant.

Acknowledgements

We would like to thank Karel Vesely, who wrote the original "nnet1" neural network training code upon which the work here is based; Hagen Soltau and Oriol Vinyals for fruitful discussions; Ehsan Variani and Pegah Ghahremani for their work on CUDA kernels; and as many others, too numerous to name, who have contributed to some aspect of the neural net setup and to Kaldi more generally.

The authors were supported by DARPA BOLT contract No HR0011-12-C-0015, and IARPA BABEL contract No W911NF-12-C-0015. We gratefully acknowledge the support of Cisco Systems, inc. (grant #574560) and Google, Inc. (Award 2012_R2_106, "Deep Neural Networks for Speech Recognition"), funds which were used to buy computer equipment and cloud computing time that were used in the development of these methods.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, IARPA, DoD/ARL or the U.S. Government.

References

[1] Frank Seide, Gang Li, and Dong Yu, "Conversational speech transcription using context-dependent deep neural networks.," in *INTERSPEECH*, 2011, pp. 437–440.

[2] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," *arXiv preprint arXiv:1106.5730*, 2011.

[3] D. Povey, A. Ghoshal, et al., "The Kaldi Speech Recognition Toolkit," in *Proc. ASRU*, 2011.

[4] Andrew Senior, Georg Heigold, Marc'Aurelio Ranzato, and Ke Yang, "An empirical study of learning rates in deep neural networks for speech recognition," in *Acoustics, Speech and Signal Processing (ICASSP)*, 2013.

[5] Howard Hua Yang and Shun-ichi Amari, "Complexity issues in natural gradient descent method for training multilayer perceptrons," *Neural Computation*, vol. 10, no. 8, pp. 2137–2157, 1998.

[6] Nicolas Le Roux, Yoshua Bengio, and Pierre antoine Manzagol, "Topmoumoute online natural gradient algorithm," in *NIPS*, 2007.

[7] Léon Bottou, "Online learning and stochastic approximations," *On-line learning in neural networks*, vol. 17, pp. 9, 1998.

[8] Shun-Ichi Amari, "Natural gradient works efficiently in learning," *Neural Computation*, vol. 10, pp. 251–276, 1998.

[9] Noboru Murata and Shun-ichi Amari, "Statistical analysis of learning dynamics," *Signal Processing*, vol. 74, no. 1, pp. 3–28, 1999.

[10] Michael R Bastian, Jacob H Gunther, and Todd K Moon, "A simplified natural gradient learning algorithm," *Advances in Artificial Neural Systems*, vol. 2011, pp. 3, 2011.

[11] Howard Hua Yang and Shun-ichi Amari, "Natural gradient descent for training multi-layer perceptrons," *Unpublished (submitted to IEEE Tr. on Neural Networks)*, 1997.

[12] Razvan Pascanu and Yoshua Bengio, "Natural gradient revisited," *CoRR*, vol. abs/1301.3584, 2013.

[13] Gianna M Del Corso, "Estimating an eigenvector by the power method with a random start," *SIAM Journal on Matrix Analysis and Applications*, vol. 18, no. 4, pp. 913–937, 1997.

[14] Xiaohui Zhang, Jan Trmal, Daniel Povey, and Sanjeev Khudanpur, "Improving deep neural network acoustic models using generalized maxout networks," in *Proc. ICASSP*, 2014.

[15] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio, "Maxout networks," *arXiv preprint arXiv:1302.4389*, 2013.

[16] Harold J Kushner and George Yin, *Stochastic approximation and recursive algorithms and applications*, vol. 35, Springer, 2003.

[17] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller, "Efficient backprop," in *Neural networks: Tricks of the trade*, pp. 9–48. Springer, 2012.

[18] Geoffrey Hinton, Simon Osindero, and Yee-Whye Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[19] Frank Seide, Gang Li, Xie Chen, and Dong Yu, "Feature engineering in context-dependent deep neural networks for conversational speech transcription," in *Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on*. IEEE, 2011, pp. 24–29.

[20] Xavier Glorot and Yoshua Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *International Conference on Artificial Intelligence and Statistics*, 2010, pp. 249–256.

[21] Abdel-rahman Mohamed, Dong Yu, and Li Deng, "Investigation of full-sequence training of deep belief networks for speech recognition.," in *INTERSPEECH*, 2010, pp. 2846–2849.

[22] D. Povey, *Discriminative Training for Large Voculabulary Speech Recognition*, Ph.D. thesis, Cambridge University, 2004.

[23] L Brown Bahl, P de Souza, and R P Mercer, "Maximum mutual information estimation of hidden markov model parameters for speech recognition," Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'86., 1986.

[24] D. Povey and D. Kanevsky and B. Kingsbury and B. Ramabhadran and G. Saon and K. Visweswariah, "Boosted MMI for Feature and Model Space Discriminative Training," in *ICASSP*, 2008.

[25] D. Povey. and P. C. Woodland, "Minimum Phone Error and I-smoothing for Improved Discriminative Training," in *ICASSP*, 2002.

[26] Gibson M. and Hain T., "Hypothesis Spaces For Minimum Bayes Risk Training In Large Vocabulary Speech Recognition," in *Interspeech*, 2006.

[27] Daniel Povey and Brian Kingsbury, "Evaluation of proposed modifications to MPE for large scale discriminative training," in *ICASSP*, 2007.

[28] Karel Veselỳ, Arnab Ghoshal, Lukáš Burget, and Daniel Povey, "Sequence-discriminative training of deep neural networks," in *Proc. Interspeech*, 2013.

[29] M. J. F. Gales and P. C. Woodland, "Mean and Variance Adaptation Within the MLLR Framework," *Computer Speech and Language*, vol. 10, pp. 249–264, 1996.

[30] Najim Dehak, Patrick Kenny, Réda Dehak, Pierre Dumouchel, and Pierre Ouellet, "Front-end factor analysis for speaker verification," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 19, no. 4, pp. 788–798, 2011.

[31] Steven Davis and Paul Mermelstein, "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 28, no. 4, pp. 357–366, 1980.

[32] Andrew Senior and Ignacio Lopez-Moreno, "Improving dnn speaker independence with i-vector inputs," in *Proc. ICASSP*, 2014.

A Further details on simple natural gradient method

A.1 Overview of simple natural gradient method

In this section we describe the natural gradient method that uses the other elements of the minibatch to estimate the factors of the Fisher matrix.

As mentioned in Section 4.7, the interface can be described as follows. Given a matrix $\mathbf{X}$, each row of which represents one element of the minibatch (and with a number of columns corresponding to either the row or column dimension of one of the weight matrices in the network), do the inverse-Fisher multiplication for each row $\mathbf{x}_i$ of $\mathbf{X}$ and return the modified matrix $\bar{\mathbf{X}}$.

The core of the inverse-Fisher multiplication is this: let $\bar{\mathbf{x}}_i = \mathbf{F}_i^{-1}\mathbf{x}_i$, where $\mathbf{F}_i$ is the Fisher matrix estimated from the *other* rows of $\mathbf{X}$, i.e. if N is the minibatch size, then $\mathbf{F}_i = \frac{1}{N-1}\sum_{j\neq i}\mathbf{x}_j\mathbf{x}_j^T$. We extend this basic idea by adding smoothing of $\mathbf{F}_i$ with the identity matrix, and by scaling the output $\bar{\mathbf{X}}$ to have the same Frobenius norm as the input.

A.2 Details of method (not considering efficiency)

In this section we describe what we compute in our "simple" natural gradient method, without considering how to compute it efficiently. As described in Section 4.9, we smooth the Fisher matrix with the identity. Defining

$$\beta = \alpha \max(\mathrm{tr}\,(\mathbf{X}^T\mathbf{X}), \epsilon)/(ND) \tag{8}$$

where our normal settings are $\alpha = 4$ and $\epsilon = 10^{-20}$, and N and D are the number of rows and columns respectively of $\mathbf{X}$, we define smoothed Fisher matrices as follows, to be applied to each row $\mathbf{x}_i$ of $\mathbf{X}$:

$$\mathbf{G}_i = \left(\beta\mathbf{I} + \frac{1}{N-1}\sum_{j\neq i}\mathbf{x}_j\mathbf{x}_j^T\right)^{-1} \tag{9}$$

For each row i we will then define

$$\hat{\mathbf{x}}_i = \mathbf{G}_i^{-1}\mathbf{x}_i \tag{10}$$

and then the result of our computation will be

$$\bar{\mathbf{X}} = \gamma\hat{\mathbf{X}} \tag{11}$$

where the rescaling factor γ, intended to make sure that $\bar{\mathbf{X}}$ has the same Frobenius norm as the input $\mathbf{X}$, is defined as

$$\gamma = \sqrt{\mathrm{tr}\,(\mathbf{X}^T\mathbf{X})/\mathrm{tr}\,(\hat{\mathbf{X}}^T\hat{\mathbf{X}})}. \tag{12}$$

If the denominator of the above is zero, we take γ to be one.

We should note that by computing the scalars β and γ without "holding out" the current sample, we are violating the rule that the randomly sampled learning rate matrix should be independent of the current sample. However, since we always use a fairly large minibatch size (at least 100) and these are only scalar quantities, we don't believe the small amount of "contamination" that takes place here will significantly bias the training. In fact, it might not turn out to be very difficult to modify the equations to properly hold out the current sample for these purposes, but because we don't believe it would perceptibly affect the results, we haven't gone to the trouble of doing this.

A.3 Efficient computation in simple method

We now describe how we efficiently compute what we described above. Define the smoothed Fisher matrix

$$\mathbf{G} = \left(\beta\,\mathbf{I} + \tfrac{1}{N-1}\mathbf{X}^T\mathbf{X}\right), \tag{13}$$

which is like the $\mathbf{G}_i$ quantities but without holding out the current sample. Next, compute

$$\mathbf{Q} = \mathbf{X}\mathbf{G}^{-1}, \tag{14}$$

where $\mathbf{Q}$ only differs from $\hat{\mathbf{X}}$ by $\mathbf{x}_i$ not being held out from the corresonding $\mathbf{G}_i$. There are two equivalent methods to compute $\mathbf{Q}$:

(i) In column space:

$$\mathbf{Q} = \mathbf{X}\left(\beta\mathbf{I} + \tfrac{1}{(N-1)}\mathbf{X}^T\mathbf{X}\right)^{-1}$$

(ii) In row space:

$$\mathbf{Q} = \left(\beta\mathbf{I} + \tfrac{1}{(N-1)}\mathbf{X}\mathbf{X}^T\right)^{-1}\mathbf{X}$$

We derived the rather surprising row-space version of the formulation by expanding the inverted expression on the right of the column-space expression using the Morrison-Woodbury formula, and simplifying the resulting expression.

For efficiency, we choose method (i) above if the minibatch size is greater than the dimension ($N > D$), and method (ii) otherwise. Our formula below for $\hat{\mathbf{X}}$ is derived by expressing each $\mathbf{G}_i^{-1}$ as a rank-one correction to $\mathbf{G}^{-1}$, and computing the corresponding correction by which each row $\hat{\mathbf{x}}_i$ differs from the corresponding row $\mathbf{q}_i$ of $\mathbf{Q}$. It turns out that the correction is in the same direction as $\mathbf{q}_i$ itself, so $\hat{\mathbf{x}}_i$ just becomes a scalar multiple of $\mathbf{q}_i$. Defining, for each row-index i,

$$a_i = \mathbf{x}_i^T\mathbf{q}_i, \tag{15}$$

and defining the scalar factor

$$b_i = 1 + a_i/(N-1-a_i), \tag{16}$$

then we can use the following efficient formula: for each row $\hat{\mathbf{x}}_i$ of $\hat{\mathbf{X}}$,

$$\hat{\mathbf{x}}_i = b_i \mathbf{q}_i. \tag{17}$$

We then get the output $\bar{\mathbf{X}}$ by scaling $\hat{\mathbf{X}}$ by γ, as described above.

When working on CPUs with small minibatch sizes (e.g. $N = 128$) and large hidden-layer dimensions (e.g. $D = 1000$), the computation above is very efficient, and does not comprise more than about 20% of the time of the overall backprop computation. However, when using GPUs with larger minibatch sizes (e.g. $N = 512$) it can take the majority of the time. Even though it typically takes considerably less than half of the total floating point operations of the overall computation, it contains a matrix inversion, and matrix inversions are not very easy to compute on a GPU. Our "online" method which we will describe below is designed to solve this efficiency problem.

B Further details on online natural gradient method

B.1 Overview of online natural gradient method

The interface of the online natural-gradient method is essentially the same as the simple method: the user provides a matrix $\mathbf{X}$, and we return a matrix $\bar{\mathbf{X}}$ that's been multiplied by the inverse-Fisher and then rescaled to have the same Frobenius norm as $\mathbf{X}$. Again, each row of $\mathbf{X}$ corresponds to an element of the minibatch and the column dimension corresponds to the row or column dimension of one of the weight matrices. A difference from the simple method is that the online method is "stateful", because we maintain a running estimate of the Fisher matrix. Each time we process a minibatch, we use the Fisher matrix estimated from the previous minibatches; and we then update that estimate using the current minibatch. For a single neural net, the number of separate copies of this "state" that we need to maintain corresponds to twice the number of trainable weight matrices in the neural net: one for each of the $\mathbf{A}_i$ and $\mathbf{B}_i$ quantities in Equation (2).

Let the input be $\mathbf{X} \in \mathbb{R}^{N \times D}$, where N is the minibatch size (e.g. 512) and D is the row or column size of the weight matrix we're updating (e.g. 2000). We introduce a user-specified parameter $R < D$ which is the rank of the non-identity part of the Fisher matrix. Let the subscript $t = 0, 1, \ldots$ correspond to the minibatch. Define

$$\mathbf{F}_t \stackrel{def}{=} \mathbf{R}_t^T \mathbf{D}_t \mathbf{R}_t + \rho_t \mathbf{I} \tag{18}$$

where $\mathbf{R}_t \in \mathbb{R}^{R \times D}$, $\mathbf{D}_t \in \mathbb{R}^{R \times R}$ and $\rho_t > 0$ will be estimated online from data; $\mathbf{R}_t$ has orthonormal rows and $\mathbf{D}_t$ is diagonal and nonnegative. We'll estimate these quantities online from the data with the aim being that $\mathbf{F}_t$ should be a good estimate of the covariance of the rows of the $\mathbf{X}_t$ quantities.

We define $\mathbf{G}_t$ to be a kind of "smoothed" version of $\mathbf{F}_t$ where we add in more of the unit matrix, controlled by the α parameter we've previously discussed (normally $\alpha = 4$):

$$\mathbf{G}_t \stackrel{def}{=} \mathbf{F}_t + \frac{\alpha \mathrm{tr}\,(\mathbf{F}_t)}{D} \mathbf{I}. \tag{19}$$

and then the output will be:

$$\bar{\mathbf{X}}_t = \gamma_t \mathbf{X}_t \mathbf{G}_t^{-1} \tag{20}$$

where γ_t is computed so as to ensure that the Frobenius norm of $\bar{\mathbf{X}}_t$ equals that of $\mathbf{X}_t$:

$$\gamma_t = \sqrt{\mathrm{tr}\,(\mathbf{X}_t \mathbf{X}_t^T)/\mathrm{tr}\,(\mathbf{X}_t \mathbf{G}_t^{-1} \mathbf{G}_t^{-1} \mathbf{X}_t^T)}, \tag{21}$$

or $\gamma_t = 1$ if the denominator of the above equation is 0.

B.2 Updating our low-rank approximation to the variance

Next we discuss the method we use to estimate our low-rank approximation $\mathbf{F}_t$ of the uncentered covariance of the rows of the inputs $\mathbf{X}_t$. Define

$$\mathbf{S}_t \stackrel{def}{=} \frac{1}{N} \mathbf{X}_t^T \mathbf{X}_t \tag{22}$$

as the uncentered covariance of the rows of $\mathbf{X}_t$. We introduce a user-specified "forgetting factor" $0 < \eta < 1$ (we describe how this is set in Section B.4), and we define

$$\mathbf{T}_t \stackrel{def}{=} \eta \mathbf{S}_t + (1 - \eta) \mathbf{F}_t. \tag{23}$$

We will try to set $\mathbf{F}_{t+1}$ to be a good low-rank approximation to $\mathbf{T}_t$. The obvious way would be to make $\mathbf{D}_{t+1}$ correspond to the top eigenvalues of $\mathbf{T}_t$ and $\mathbf{R}_{t+1}$ to the corresponding eigenvectors, but this would be too slow. Instead we use a method inspired by the power method for finding the top eigenvalue of a matrix. On each iteration we compute

$$\mathbf{Y}_t \stackrel{def}{=} \mathbf{R}_t \mathbf{T}_t, \tag{24}$$

with $\mathbf{Y}_t \in \mathbb{R}^{R \times D}$. It is useful to think of $\mathbf{Y}_t$ as containing each eigenvector scaled by its corresponding eigenvalue in $\mathbf{T}_t$ (of course, this is true in a precise sense only at convergence). Our update uses symmetric eigenvalue decomposition of $\mathbf{Y}_t \mathbf{Y}_t^T$ to find these

scaling factors (they are actually the square roots of the eigenvalues of $\mathbf{Y}_t\mathbf{Y}_t^T$), puts them on the diagonal of $\mathbf{D}_{t+1}$, and puts the corresponding eigenvectors in the rows of $\mathbf{R}_{t+1}$. We then have to work out the correct amount of the unit-matrix to add into our factorization of the covariance matrix (i.e. set ρ_{t+1}) and subtract that amount from the diagonals of $\mathbf{D}_{t+1}$. We will give equations for this below.

Observant readers might have noted that it would seem more straightforward do do a Singular Value Decomposition (SVD) on $\mathbf{Y}_t$ instead of a symmetric eigenvalue decomposition on $\mathbf{Y}_t\mathbf{Y}_t^T$. We do it this way for speed.

The details of our update are as follows:

$$\mathbf{Z}_t \stackrel{def}{=} \mathbf{Y}_t\mathbf{Y}_t^T, \tag{25}$$

so $\mathbf{Z}_t \in \mathbb{R}^{R\times R}$. Then do the symmetric eigenvalue decomposition

$$\mathbf{Z}_t = \mathbf{U}_t\mathbf{C}_t\mathbf{U}_t^T, \tag{26}$$

with $\mathbf{U}$ orthogonal and $\mathbf{C}_t$ diagonal. The diagonal elements of $\mathbf{C}_t$ are positive; we can prove this using $\rho_t > 0$ (which makes $\mathbf{T}_t$ positive definite) and using the fact that $\mathbf{R}_t$ has full row rank. We define $\mathbf{R}_{t+1}$ as:

$$\mathbf{R}_{t+1} \stackrel{def}{=} \mathbf{C}_t^{-0.5}\mathbf{U}_t^T\mathbf{Y}_t \tag{27}$$

If we expand out $\mathbf{R}_{t+1}\mathbf{R}_{t+1}^T$ using (27), it is easy to see that it reduces to the identity, hence $\mathbf{R}_{t+1}$ has orthonormal rows. In order to make sure that $\mathbf{F}_{t+1}$ has the desired covariance in the directions corresponding to the rows of $\mathbf{R}_{t+1}$, we will set

$$\mathbf{D}_{t+1} \stackrel{def}{=} \mathbf{C}_t^{0.5} - \rho_{t+1}\mathbf{I}, \tag{28}$$

but note that at this point, ρ_{t+1} is still unknown. When we say the "desired covariance", we are ensuring that for each dimension $\mathbf{r}$ corresponding to a row of $\mathbf{R}_{t+1}$, the value of the inner product $\mathbf{r}^T\mathbf{F}_{t+1}\mathbf{r}$ equals that of $\mathbf{r}^T\mathbf{T}_t\mathbf{r}$, but this is only precisely true at convergence.

We choose ρ_{t+1} in order to ensure that $\mathrm{tr}\,(\mathbf{F}_{t+1}) = tr(\mathbf{T}_t)$. This value can be worked out as:

$$\rho'_{t+1} = \tfrac{1}{D-R}\left(\eta\mathrm{tr}\,(\mathbf{S}_t)+(1-\eta)(D\rho_t+\mathrm{tr}\,(\mathbf{D}_t))-\mathrm{tr}\,(\mathbf{C}_t^{0.5})\right) \tag{29}$$

We then let

$$\rho_{t+1} = \max(\epsilon, \rho'_{t+1}) \tag{30}$$

for $\epsilon = 10^{-10}$; this is is to ensure that if we get a sequence of zero inputs, ρ_t will not become exactly zero in its machine representation.

B.3 Efficient computation

The previous section described what we are computing in the online natural gradient method; here we describe how to compute it efficiently. The essential idea here is to reduce the multiplication by $\mathbf{G}^{-1}$ to two multiplications by a "fat" matrix (of dimension $R\times D$). Since typically R is much smaller than D, this is quite efficient. We also address how to efficiently keep these matrices updated, at the level of optimizing the matrix expressions. This section is mostly derivation, and will likely only be of interest to someone who is considering implementing this method. In Section B.5 below, we will summarize the algorithm we derive here.

We can write $\mathbf{G}_t$ as:

$$\mathbf{G}_t \stackrel{def}{=} \mathbf{F}_t + \tfrac{\alpha\mathrm{tr}\,(\mathbf{F}_t)}{D}\mathbf{I} \tag{31}$$

$$= \mathbf{R}_t^T\mathbf{D}_t\mathbf{R}_t + \beta_t I \tag{32}$$

where

$$\beta_t \stackrel{def}{=} \rho_t + \tfrac{\alpha}{D}\mathrm{tr}\,(\mathbf{F}_t) \tag{33}$$

$$= \rho_t(1+\alpha) + \tfrac{\alpha}{D}\mathrm{tr}\,(\mathbf{D}_t) \tag{34}$$

Define

$$\hat{\mathbf{X}}_t \stackrel{def}{=} \beta_t\mathbf{X}_t\mathbf{G}_t^{-1}, \tag{35}$$

where the factor of β_t is inserted arbitrarily to simplify the update equations; a scalar factor on $\hat{\mathbf{X}}$ doesn't matter because we will later rescale it to have the same norm as $\mathbf{X}$. The output of this whole process is

$$\bar{\mathbf{X}}_t \stackrel{def}{=} \gamma_t\hat{\mathbf{X}}_t, \text{ where} \tag{36}$$

$$\gamma_t \stackrel{def}{=} \sqrt{\mathrm{tr}\,(\mathbf{X}_t\mathbf{X}_t^T)/\mathrm{tr}\,(\hat{\mathbf{X}}_t^T\hat{\mathbf{X}}_t)}, \tag{37}$$

where, in the expression for γ_t, if the denominator is zero we take $\gamma_t = 1$. Note: γ_t is not the same as in (21) because of the arbitrary factor of β_t, so consider (21) to be superseded by (37). To efficiently compute (35), we apply the Woodbury matrix identity to (31), giving us

$$\mathbf{G}_t^{-1} = \frac{1}{\beta_t}\left(\mathbf{I} - \mathbf{R}_t^T\mathbf{E}_t\mathbf{R}_t\right) \tag{38}$$

where

$$\mathbf{E}_t \stackrel{def}{=} \frac{1}{\beta_t}\left(\mathbf{D}_t^{-1} + \frac{1}{\beta_t}\mathbf{I}\right)^{-1} \tag{39}$$

with elements

$$e_{tii} = \frac{1}{\beta_t/d_{tii} + 1} \tag{40}$$

In order to reduce the number of matrix multiplies, it is useful to break the expression $\mathbf{R}_t^T\mathbf{E}_t\mathbf{R}_t$ into two equal parts, so we define

$$\mathbf{W}_t \stackrel{def}{=} \mathbf{E}_t^{0.5}\mathbf{R}_t, \tag{41}$$

and we will never store $\mathbf{R}_t$; instead, we will work with $\mathbf{W}_t$ and the small diagonal factors $\mathbf{D}_t$ and $\mathbf{E}_t$. We can now write the following, which is where most of our computation will take place:

$$\hat{\mathbf{X}}_t = \mathbf{X}_t - \mathbf{X}_t \mathbf{W}_t^T \mathbf{W}_t \tag{42}$$

You may recall the symmetric matrix $\mathbf{Z}_t \in \mathbb{R}^{R \times R}$ defined in (25), which is involved in the update of our factorization. The following expressions are going to be useful when computing it, and the first of them appears as a sub-expression of (42). For convenience we state the dimensions of these quantities below:

$$\mathbf{H}_t \stackrel{def}{=} \mathbf{X}_t \mathbf{W}_t^T \in \mathbb{R}^{N \times R} \tag{43}$$

$$\mathbf{J}_t \stackrel{def}{=} \mathbf{H}_t^T \mathbf{X}_t \in \mathbb{R}^{R \times D} \tag{44}$$

$$= \mathbf{W}_t \mathbf{X}_t^T \mathbf{X}_t \tag{45}$$

$$\mathbf{K}_t \stackrel{def}{=} \mathbf{J}_t \mathbf{J}_t^T \in \mathbb{R}^{R \times R} (\text{symmetric}) \tag{46}$$

$$\mathbf{L}_t \stackrel{def}{=} \mathbf{H}_t^T \mathbf{H}_t \in \mathbb{R}^{R \times R} (\text{symmetric}) \tag{47}$$

$$= \mathbf{W}_t \mathbf{X}_t^T \mathbf{X}_t \mathbf{W}_t^T \tag{48}$$

$$= \mathbf{J}_t \mathbf{W}_t^T \tag{49}$$

After we have $\mathbf{H}_t$, we can compute $\hat{\mathbf{X}}_t$ using a single matrix multiply as:

$$\hat{\mathbf{X}}_t = \mathbf{X}_t - \mathbf{H}_t \mathbf{W}_t. \tag{50}$$

We can expand $\mathbf{Y}_t = \mathbf{R}_t \mathbf{T}_t$, defined in (24), into quantities that will be computed, as:

$$\mathbf{Y}_t = \tfrac{\eta}{N} \mathbf{R}_t \mathbf{X}_t^T \mathbf{X}_t + (1-\eta)(\mathbf{D}_t + \rho_t \mathbf{I}) \mathbf{R}_t \tag{51}$$

$$= \tfrac{\eta}{N} \mathbf{E}_t^{-0.5} \mathbf{J}_t + (1-\eta)(\mathbf{D}_t + \rho_t \mathbf{I}) \mathbf{E}_t^{-0.5} \mathbf{W}_t \tag{52}$$

Using (52) we can expand $\mathbf{Z}_t = \mathbf{Y}_t \mathbf{Y}_t^T$, as:

$$\begin{aligned} \mathbf{Z}_t = {} & \tfrac{\eta^2}{N^2} \mathbf{E}_t^{-0.5} \mathbf{J}_t \mathbf{J}_t^T \mathbf{E}_t^{-0.5} + (1-\eta)^2 (\mathbf{D}_t + \rho_t \mathbf{I})^2 \\ & + \tfrac{\eta(1-\eta)}{N} \mathbf{E}_t^{-0.5} \mathbf{J}_t \mathbf{W}_t^T \mathbf{E}_t^{-0.5} (\mathbf{D}_t + \rho_t \mathbf{I}) \\ & + \tfrac{\eta(1-\eta)}{N} (\mathbf{D}_t + \rho_t \mathbf{I}) \mathbf{E}_t^{-0.5} \mathbf{W}_t \mathbf{J}_t^T \mathbf{E}_t^{-0.5} \end{aligned} \tag{53}$$

and we can substitute some of the sub-expressions we defined above into this, to give:

$$\begin{aligned} \mathbf{Z}_t = {} & \tfrac{\eta^2}{N^2} \mathbf{E}_t^{-0.5} \mathbf{K}_t \mathbf{E}_t^{-0.5} + (1-\eta)^2 (\mathbf{D}_t + \rho_t \mathbf{I})^2 \\ & + \tfrac{\eta(1-\eta)}{N} \mathbf{E}_t^{-0.5} \mathbf{L}_t \mathbf{E}_t^{-0.5} (\mathbf{D}_t + \rho_t \mathbf{I}) \\ & + \tfrac{\eta(1-\eta)}{N} (\mathbf{D}_t + \rho_t \mathbf{I}) \mathbf{E}_t^{-0.5} \mathbf{L}_t \mathbf{E}_t^{-0.5} \end{aligned} \tag{54}$$

Our strategy will be to compute the symmetric quantities $\mathbf{L}_t$ and $\mathbf{K}_t$ on the GPU, and transfer them to the CPU where we can then compute $\mathbf{Z}_t$ using the expression above – this can be done in $O(R^2)$ – and then do the symmetric eigenvalue decomposition as in (26), on the CPU. We repeat the equation here for convenience:

$$\mathbf{Z}_t = \mathbf{U}_t \mathbf{C}_t \mathbf{U}_t^T. \tag{55}$$

Here, $\mathbf{U}_t$ will be orthogonal, and mathematically, no element of the diagonal matrix $\mathbf{C}_t$ can be less than $(1-\eta)^2 \rho_t^2$, so we floor its diagonal to that value to prevent problems later if, due to roundoff, any element is smaller than that.

Below, we'll say how we efficiently compute $\mathrm{tr}\,(\mathbf{X}\mathbf{X}^T)$ and $\mathrm{tr}\,(\hat{\mathbf{X}}\hat{\mathbf{X}}^T)$; for now, just assume those quantities have been computed.

We compute ρ_{t+1} as follows, expanding $\mathbf{S}_t$ in (29):

$$\begin{aligned} \rho'_{t+1} = \frac{1}{D-R} \Big(& \frac{\eta}{N} \mathrm{tr}\,(\mathbf{X}\mathbf{X}^T) + \\ & (1-\eta)(D\rho_t + \mathrm{tr}\,(\mathbf{D}_t)) - \mathrm{tr}\,(\mathbf{C}_t^{0.5}) \Big). \end{aligned} \tag{56}$$

We can now compute $\mathbf{D}_{t+1}$ and ρ_{t+1}; we floor both to ϵ to ensure they never go to exactly zero which could cause problems for our algorithm.

$$\mathbf{D}_{t+1} = \max(\mathbf{C}_t^{0.5} - \rho'_{t+1} \mathbf{I}, \epsilon \mathbf{I}) \tag{57}$$

$$\rho_{t+1} = \max(\epsilon, \rho'_{t+1}) \tag{58}$$

for a small constant $\epsilon = 10^{-10}$ (the first max is taken per element). We can now compute the scalar β_{t+1} and the diagonal matrix $\mathbf{E}_{t+1}$ (we show the formula for its diagonal elements):

$$\beta_{t+1} = \rho_{t+1}(1+\alpha) + \frac{\alpha}{D} \mathrm{tr}\,(\mathbf{D}_{t+1}) \tag{59}$$

$$e_{tii} = \frac{1}{\beta_{t+1}/d_{t+1,ii} + 1} \tag{60}$$

We never construct $\mathbf{R}_{t+1}$ in memory, but instead we directly compute $\mathbf{W}_{t+1}$. We can factor it as follows:

$$\mathbf{W}_{t+1} \stackrel{def}{=} \mathbf{E}_{t+1}^{0.5} \mathbf{R}_{t+1} \tag{61}$$

$$= \mathbf{E}_{t+1}^{0.5} \mathbf{C}_t^{-0.5} \mathbf{U}_t^T \mathbf{Y}_t \tag{62}$$

$$= \mathbf{E}_{t+1}^{0.5} \mathbf{C}_t^{-0.5} \mathbf{U}_t^T \left(\tfrac{\eta}{N} \mathbf{E}_t^{-0.5} \mathbf{J}_t + (1-\eta)(\mathbf{D}_t + \rho_t \mathbf{I}) \mathbf{R}_t \right) \tag{63}$$

$$= \mathbf{A}_t \mathbf{B}_t \tag{64}$$

where

$$\mathbf{A}_t \stackrel{def}{=} \tfrac{\eta}{N} \mathbf{E}_{t+1}^{0.5} \mathbf{C}_t^{-0.5} \mathbf{U}_t^T \mathbf{E}_t^{-0.5} \tag{65}$$

$$\mathbf{B}_t \stackrel{def}{=} \mathbf{J}_t + \tfrac{N(1-\eta)}{\eta} (\mathbf{D}_t + \rho_t \mathbf{I}) \mathbf{W}_t, \tag{66}$$

and note that while it might seem like a factor of $\mathbf{E}_t^{-0.5}$ is missing from the second term in $\mathbf{B}_t$, in fact we use the fact that it commutes with $(\mathbf{D}_t + \rho_t \mathbf{I})$ to move it to the left, into $\mathbf{A}_t$. If we're using a GPU, $\mathbf{A}_t$ will be computed in time $O(R^2)$ on the CPU and transferred to the GPU; we then compute $\mathbf{B}_t$ on the GPU efficiently by scaling the rows of $\mathbf{W}_t$ and adding $\mathbf{J}_t$; then we multiply $\mathbf{A}_t$ and $\mathbf{B}_t$ on the GPU.

B.3.1 Maintaining orthogonality

We have noticed that the invariance $\mathbf{R}_t\mathbf{R}_t^T = \mathbf{I}$ can sometimes be lost due to roundoff. A proper analysis of roundoff in our algorithm is not something we have time to do, but we will describe how we detect and fix this problem in practice. For speed, we only do the following operations if the diagonal matrix $\mathbf{C}_t$, has condition number greater than 10^6, or if any elements were floored as mentioned just after (55). Note: all the computations we describe in this paper were done in single precision.

We compute the symmetric matrix

$$\mathbf{O}_t \stackrel{def}{=} \mathbf{R}_t\mathbf{R}_t^T \tag{67}$$

$$= \mathbf{E}_t^{-0.5}\left(\mathbf{W}_t\mathbf{W}_t^T\right)\mathbf{E}_t^{-0.5}, \tag{68}$$

where the part in parentheses is computed on the GPU and transferred to the CPU. If no element of $\mathbf{O}_t$ differs by more than 10^{-3} from the corresponding element of the unit matrix, we consider that $\mathbf{R}_t$ is sufficiently orthogonal and we do nothing more. Otherwise, we do a Cholesky decomposition $\mathbf{O}_t = \mathbf{C}\mathbf{C}^T$, compute the reorthogonalizing factor $\mathbf{M} = \mathbf{E}_t^{0.5}\mathbf{C}^{-1}\mathbf{E}_t^{-0.5}$ on the CPU and copy to the GPU, and do $\mathbf{W}_{t+1} \leftarrow \mathbf{M}\mathbf{W}_{t+1}$ to reorthogonalize. Re-orthogonalization happens extremely rarely, and usually only if something bad has already happened such as parameter divergence.

B.3.2 Initialization

In our implementation we don't bother dumping the "state' of the computation to disk so each new process reinitializes them for the first minibatch it processes. We initialize them so as to most closely approximate the covariance of the first minibatch of features. This is done by taking

$$\mathbf{S}_0 \stackrel{def}{=} \tfrac{1}{N}\mathbf{X}_0^T\mathbf{X}_0 \tag{69}$$

and finding the top R eigenvalues and eigenvectors; the rows of $\mathbf{R}_0$ contain the top eigenvectors. Let λ_i be the corresponding eigenvalues, for $1 \le i \le R$, and we set

$$\rho_0 = \max\left(\frac{\mathrm{tr}\left(\mathbf{S}_0\right) - \sum_{i=1}^R \lambda_i}{D - R}, \epsilon\right) \tag{70}$$

for $\epsilon = 10^{-10}$, and for $1 \le i \le R$, we let $d_{0ii} \leftarrow \max(\epsilon, \lambda_i - \rho_0)$.

B.3.3 Computing matrix traces

We mentioned above that we have a fast way of computing the quantities $\mathrm{tr}\left(\mathbf{X}\mathbf{X}^T\right)$ and $\mathrm{tr}\left(\hat{\mathbf{X}}\hat{\mathbf{X}}^T\right)$. These are needed to compute γ_t using (37), and to compute ρ'_{t+1} using (56). We compute these as a side effect of the fact that we need, for each row $\bar{\mathbf{x}}_{ti}$ of the output, its squared norm $\bar{\mathbf{x}}_{ti}^T\bar{\mathbf{x}}_{ti}$. This will be required to enforce the "maximum parameter change" per minibatch, as described in Section 3.4.2. Suppose we've already computed $\hat{\mathbf{X}}_t$ using (50). We compute the inner products for all rows $1 \le i \le N$ of $\hat{\mathbf{X}}_t$ as

$$p_i = \hat{\mathbf{x}}_{ti}^T\hat{\mathbf{x}}_{ti}^T, \tag{71}$$

using a single GPU kernel invocation. If we are updating the parameters of the Fisher-matrix factorization, then we can most efficiently obtain our desired traces as follows:

$$\mathrm{tr}\left(\hat{\mathbf{X}}\hat{\mathbf{X}}^T\right) = \textstyle\sum_i p_i \tag{72}$$

$$\mathrm{tr}\left(\mathbf{X}\mathbf{X}^T\right) = \mathrm{tr}\left(\hat{\mathbf{X}}\hat{\mathbf{X}}^T\right) - \mathrm{tr}\left(\mathbf{L}_t\mathbf{E}_t\right) + 2\mathrm{tr}\left(\mathbf{L}_t\right). \tag{73}$$

The expression for $\mathrm{tr}\left(\mathbf{X}\mathbf{X}^T\right)$ was obtained by expanding $\mathrm{tr}\left(\hat{\mathbf{X}}\hat{\mathbf{X}}^T\right)$ using (50), moving $\mathrm{tr}\left(\mathbf{X}\mathbf{X}^T\right)$ to the left, and recognizing sub-expressions that we have already computed. In case we are not updating the parameters of the Fisher-matrix factorization, we have no other need for $\mathbf{L}_t$ so it will be more efficient to compute $\mathrm{tr}\left(\mathbf{X}\mathbf{X}^T\right)$ directly; this can of course be done in $O(ND)$ operations and does not require a matrix multiply. Once we have the scaling factor γ_t we can scale the p_i quantities by its square, and they will equal the quantities $\bar{\mathbf{x}}_{ti}^T\bar{\mathbf{x}}_{ti}$ that we'll need for enforcing the maximum parameter change.

B.3.4 Multithreading and other issues

Most of what we have written above is geared towards operation using a GPU, but we also support operation with CPUs, where our SGD implementation is multithreaded. In this case, we have to consider the interaction with multithreaded code because of the "stateful" nature of the computation. We wanted to avoid a bottleneck where different threads wait to update the parameters sequentially. Our solution is that before doing the part of the computation where we update the parameters, we try to get a lock, and if this fails, we simply apply the fixed preconditioning matrix but don't update the Fisher-matrix parameters $\mathbf{R}_t$ and so on. Since the model parameters don't move very fast, we don't expect that this will make any noticeable difference to the SGD convergence, and we have seen no evidence that it does.

B.4 Typical configuration

The most important user-specified parameters for our algorithm are the rank R and the constant α that controls smoothing with the unit matrix. The value $\alpha = 4$ seems to work well over a wide variety of conditions, so we normally leave it at that value. The rank R

should generally increase with the dimension of the vectors we are multiplying. Our experiments here are with "p-norm" networks where the nonlinearity is dimension reducing, like maxout [15], typically reducing the dimension from something like 3000 to 300. So a typical parameter matrix will increase the dimension from something like 301 to 3000 (it's 301 instead of 300 because of the bias term). Our normal rule for ranks is to use $R = 20$ on the input side of each matrix and $R = 80$ on the output side. Part of the way we originally tuned this is to look at the diagonal matrices $\mathbf{E}_t$. These matrices have diagonal values $0 < e_{tii} < 1$, sorted on i from greatest to least, and $1 - e_{tii}$ can be interpreted as the amount by which the input is scaled in a certain direction in the space. A value of e_{tii} close to 1 means we are strongly scaling down the input, and a value close to 0 means we are leaving it unchanged. If the last e_{tii} has a value of, say, 0.1, then reducing R by one will be like taking a scaling factor of 0.9 applied to a gradient, and setting to 1 instead; this seems unlikely to make any difference to the SGD, as it's like changing the learning rate in some direction from 0.9 to 1. Our final e_{tii} values are normally in the range 0.05 to 0.2.

Another configurable constant is the "forgetting factor" $0 < \eta < 1$: the closer η is to 1, the more rapidly we track changes in the Fisher matrix due to changes in parameters, but the more noise we will have in our estimates. Because we don't want to have to tune η when we change the minibatch size, we set it as follows. The user specifies a parameter S (interpreted as an approximate number of samples to include in our estimate of the Fisher matrix), and we set

$$\eta = 1 - \exp(-N/S), \tag{74}$$

where N is the minibatch size. We normally set $S = 2000$; we have no reason to believe that this is a very important parameter.

In order to increase the speed of the algorithm, we normally configure it so that we only actually update the parameters of the Fisher matrix every 4 minibatches, except on the first 10 minibatches in a process, when we always update them.

B.5 Summary of the online natural gradient method

Here we summarize the online natural-gradient SGD method– that is, we summarize the core part of the algorithm that takes a matrix $\mathbf{X} \in \mathbb{R}^{N \times D}$, and outputs a matrix $\bar{\mathbf{X}} \in \mathbb{R}^{N \times D}$. To understand how this fits into the bigger picture of back-propagation and SGD, see Section 4.

For this summary we will ignore issues of multithreading. Our explanation here is just for one instance of the algorithm, corresponding to the row or column dimension of one of the weight matrices; if there are I weight matrices, there are $2I$ separate copies of the variables we describe here.

Typical configuration variables are as follows: $\alpha = 4$, $S = 2000$ (this will determine η), rank $R = 20$ (or 80), $\epsilon = 10^{-10}$; and let's define a variable $J = 4$ that dictates the period with which we update the Fisher-matrix factors. Minibatch size N is normally 128 or 512.

On $t = 0$, before running the steps below we have to initialize the parameters as described in Section B.3.2. Note: while in Section B.3.2 we describe how to set ρ_0, $\mathbf{R}_0$ and $\mathbf{D}_0$, the variables which we actually store are ρ_0, $\mathbf{D}_0$, and $\mathbf{W}_0$; to compute $\mathbf{W}_0$ we need Equations (34), (40) and (41).

We have an input $\mathbf{X} \in \mathbb{R}^{N \times D}$, and despite the notation, we do not require that N be the same for all t– sometimes the last minibatch we process has a smaller than normal size.

If $t < 10$ or J divides t exactly, then we will be updating the factored Fisher matrix; otherwise we just apply it and don't update. There are two slightly versions of the algorithm, depending whether we will be updating the Fisher matrix.

In either case, we first compute η from N and S using (74), and then compute

$$\mathbf{H}_t = \mathbf{X}_t \mathbf{W}_t^T. \tag{75}$$

From here the two cases begin to differ.

Without updating the Fisher matrix. If we won't be updating the Fisher matrix, then it's simpler. The input is $\mathbf{X}_t$. We first compute $\mathrm{tr}\,(\mathbf{X}_t^T \mathbf{X})$. Then we compute

$$\hat{\mathbf{X}}_t = \mathbf{X}_t - \mathbf{H}_t \mathbf{W}_t, \tag{76}$$

overwriting the input $\mathbf{X}_t$. Next, for each $1 \leq i \leq N$ we compute the row-products p_i using (71), and compute $\mathrm{tr}\,(\hat{\mathbf{X}}^T \hat{\mathbf{X}})$ as the sum of p_i. Now we can compute γ_t using (37). Next we scale $\hat{\mathbf{X}}_t$ by γ_t to produce $\bar{\mathbf{X}}_t$. We also output for each i the quantity $\gamma_t^2 p_i = \bar{\mathbf{x}}_{ti}^T \bar{\mathbf{x}}_{ti}$, which is needed to enforce the "maximum parameter change per minibatch" constraint.

With updating the Fisher matrix. If we're updating the Fisher matrix, which we usually do every four steps, there are some more operations to do. First we compute

$$\mathbf{J}_t = \mathbf{H}_t^T \mathbf{X}_t \in \mathbb{R}^{R \times D}. \tag{77}$$

Next we want to compute $\mathbf{L}_t$ and $\mathbf{K}_t$. We actually have two separate strategies for this. If $N > D$ (the minibatch size exceeds the vector dimension), we do:

$$\mathbf{L}_t = \mathbf{W}_t \mathbf{J}_t^T \in \mathbb{R}^{R \times R} \tag{78}$$

$$\mathbf{K}_t = \mathbf{J}_t \mathbf{J}_t^T \in \mathbb{R}^{R \times R} \tag{79}$$

and in our implementation we combine these into one matrix operation by placing $\mathbf{L}$ and $\mathbf{K}$, and $\mathbf{W}$ and $\mathbf{J}$, next to each other in memory. Otherwise, we compute $\mathbf{K}_t$ as above but $\mathbf{L}_t$ using:

$$\mathbf{L}_t = \mathbf{H}_t^T \mathbf{H}_t \in \mathbb{R}^{R \times R}. \tag{80}$$

At this point, if we're using a GPU, we transfer the symmetric matrices $\mathbf{K}_t$ and $\mathbf{L}_t$ to the CPU. We now compute some small derived quantities on the CPU: β_t using (34) and $\mathbf{E}_t$ using (40), as well as $\mathbf{E}_t^{0.5}$ and $\mathbf{E}_t^{-0.5}$; $\mathbf{E}_t$ is diagonal so this is not hard. At this point we compute the symmetric $R \times R$ matrix $\mathbf{Z}_t$ using (54); the expression looks scary but it can be computed in $O(R^2)$ time.

We do the symmetric eigenvalue decomposition as in (55), on the CPU, to get the orthogonal matrix $\mathbf{U}_t$ and the diagonal matrix $\mathbf{C}_t$, and we floor the diagonal elements of $\mathbf{C}_t$ to $(1-\eta)^2 \rho_t^2$.

Next we compute

$$\hat{\mathbf{X}}_t = \mathbf{X}_t - \mathbf{H}_t \mathbf{W}_t, \tag{81}$$

then compute the row-products p_i using (71), compute $\mathrm{tr}\,(\hat{\mathbf{X}}_t^T \hat{\mathbf{X}}) = \sum_i p_i$, and compute $\mathrm{tr}\,(\mathbf{X}_t^T \mathbf{X})$ using (73). We can now obtain the scaling factor γ_t using (37), and use it to compute the main output $\bar{\mathbf{X}}_t = \gamma_t \hat{\mathbf{X}}_t$ and the per-row inner products of the output which equal $\gamma_t^2 p_i$ (although in our implementation, to save time we actually output γ_t and let the user do the scaling later on).

We next compute ρ'_{t+1} using (56), $\mathbf{D}_{t+1}$ using (57) and ρ_{t+1} using (58). $\mathbf{W}_{t+1}$ is computed using a matrix multiply on the GPU as in (64), after working out the factors $\mathbf{A}_t$ and $\mathbf{B}_t$.

At this point, if we had floored any diagonal elements of $\mathbf{C}_t$ above or if its condition number after flooring exceeds 10^6, we do the orthogonality check and possible reorthogonalization that we described in Section B.3.1 above.

C Other aspects of our DNN implementation

Here we describe some aspects of our neural net training implementation that are of less direct relevance to our parallel training and natural gradient methods, so were not included in the main text of the paper.

In Section C.1 we explain how we enforce a maximum parameter-change per minibatch; in C.2 we explain our generalized model-averaging procedure; in C.3 we explain how we use "mixture components" (a.k.a. subclasses) for DNNs; in C.4 we introduce our method of input data normalization; in C.5 we give details on how we initialize the DNN parameters; in C.6 we give an overview of how we implemented sequence training for DNNs; and in C.7 we discuss online (real-time) decoding using iVectors for speaker adaptation.

C.1 Enforcing the maximum parameter change per minibatch

As mentioned in Section 3.4.2, in order to prevent instability and parameter divergence we enforce a maximum parameter-change per minibatch, which is applied for each layer of the network separately. Here we explain how this is done. We don't claim that this is an exceptionally good method for preventing excessive parameter changes, but we describe it here anyway for the sake of completeness.

Suppose the update for a single weight matrix is formulated as follows (and to keep things simple, we don't include an index for the layer of the network):

$$\mathbf{W}_{t+1} = \mathbf{W}_t + \Delta_t, \tag{82}$$

where Δ_t is the change that standard SGD would give us, equal to the derivative of the objective function for this minibatch multiplied by the learning rate η_t. To enforce the maximum parameter chanbge, we scale the change by a scalar α_t:

$$\mathbf{W}_{t+1} = \mathbf{W}_t + \alpha_t \Delta_t, \tag{83}$$

where we would like to choose $\alpha_t \leq 1$ to ensure that $||\alpha_t \Delta_t||_F$ does not exceed a specified limit, $||\cdot||_F$ being the Frobenius norm. However, we don't implement this scheme exactly as described above because it would involve creating a temporary matrix to store the product of matrices Δ_t just in order to compute its norm, and we don't want to incur this penalty.

Instead we enforce it in a way that involves a sum over elements of the minibatch. If $\Delta_t = \eta \mathbf{X}^T \mathbf{Y}$, then Δ_t can be written as a sum over an index i that ranges over the rows of $\mathbf{X}$ and $\mathbf{Y}$. By properties of norms, the 2-norm of Δ_t cannot exceed the sum of the 2-norms of the terms in this sum: if the rows of $\mathbf{X}$ and $\mathbf{Y}$ are written as $\mathbf{x}_i$ and $\mathbf{y}_i$, then

$$||\Delta_t||_F \leq \sum_i \eta ||\mathbf{x}_i||_2 ||\mathbf{y}_i||_2 \tag{84}$$

It does not take excessive time or memory to compute the vector norms $||\mathbf{x}_i||_2$ and $||\mathbf{y}_i||_2$, so we compute the

right hand side of 84 and use it as a stand-in for $||\Delta_t||_F$, giving us

$$\alpha_t = \min\left(1, \frac{\text{max-change-per-minibatch}}{\sum_i \eta ||\mathbf{x}_i||_2 ||\mathbf{y}_i||_2}\right) \quad (85)$$

where max-change-per-minibatch is a user-specified maximum parameter-change per minibatch. Empirically we have found that it tends to be necessary to increase max-change-per-minibatch when using a larger minibatch size, so to simplify the configuration process we define

$$\text{max-change-per-minibatch} = N \text{max-change-per-sample} \quad (86)$$

where N is the minibatch size. We always set max-change-per-sample to 0.075 for experiments reported here. To clarify how this method interacts with the natural gradient methods described in Section 4: the natural gradient is implemented as a modification to the $\mathbf{X}$ and $\mathbf{Y}$ matrices, so we simply apply this maximum-change logic on top of the modified $\mathbf{X}$ and $\mathbf{Y}$ quantitities.

What we've found that this maximum-parameter-change limit is active only early in training for layers closer to the output.

C.2 Generalized model averaging

The convergence theory of Stochastic Gradient Descent [16] suggests that, for convex problems, if we take not the last iteration's model parameters but the average over all iterations, it can improve the convergence rate, particularly in 'poorly-conditioned' problems (i.e. where the condition number of the Hessian is very large). This is not applicable in non-convex problems such as ours, but it does suggest a related method. As mentioned above, we define an *outer iteration* as the length of time it takes for all jobs to process K samples (e.g. $K = 400\,000$), and on each outer iteration each job dumps its final model to disk and we average these to produce a single model. We store the models (averaged over all jobs) for each outer iteration. At the very end of training, instead of choosing the model from the final outer iteration, we take the models from the last P outer iterations (e.g. $P = 20$), and search for a generalized weighted combination of these models that optimizes the objective function on a subset of training data– we tried using validation data here, but for our task we found it worked best to use training data. By *generalized weighted combination*, what we mean is that the parameters are a weighted combination of the parameters of the input models, but each layer can have different weighting factors. Thus, if there are P models and L layers, the number of parameters we learn on the data subdset is LP. A few more details:

- The optimization method is L-BFGS.
- To improve the convergence speed of L-BFGS, we optimize in a transformed (preconditioned) space where the preconditioner is related to the Fisher matrix.
- The starting point for the optimization is the best of $P + 1$ choices, corresponding to each of the P final iterations, and the average of all of them.
- We add a very tiny regularizer (like 10^{-10} times the square of the vector of weights) to stop the weights going to infinity in cases (like p-norm networks) where the objective function is invariant to the parameter scale.
- We generally aim to optimize over the last $P = 20$ models (assuming they share the same parameter structure, e.g. we haven't added layers).
- In cases where P iterations would amount to less than one epoch, we optimize over P models where the individual models are simple averages of model parameters for a duration of about $1/P$ of the entire epoch.

We have generally found that this model combination slightly improves results, but it is not a focus of the current paper so we don't provide experimental results for this here.

C.3 Mixture components (sub-classes)

When using Gaussians for speech recognition, the usual approach is to use a Gaussian mixture model (GMM) rather than a single Gaussian, to model each speech state. We have generalized this idea to neural networks, by allowing the posterior of each speech state to be written as a sum over the posterior of "sub-classes" that are analogous to the Gaussians in a GMM. About halfway through training, we "mix up" the model by increasing the dimension of the softmax layer to a user-specified number that is greater than the number of classes (usually about double the number of classes). After the softmax layer we introduce a "sum-group" layer which sums its input over fixed groups of indexes to produce a posterior for each class that is a sum over the posteriors of the hidden "sub-classes". We also tried sharing the sub-classes across classes in groups, but did not find this helpful.

Rather than distributing the "sub-classes" evenly, we allocate more sub-classes to the more common classes. We allocate them proportional to the 1/3 power of the count of that class in the training data; this is based on the rule we use to allocate Gaussians in our GMMs.

When initializing the parameters of the "mixed-up" final weight matrix, we make it correspond quite closely with the original weight matrix. Each row of the new weight matrix corresponds to a row of the old weight matrix, plus a small noise term to allow the values of the rows to diverge; and we modify the bias term to normalize for the fact that some classes have more sub-classes than others.

We have generally found that this slightly improves results, but again, this is not a focus of the current paper and we won't be showing experimental results about this.

C.4 Input data normalization

As mentioned in [17, Section 4.3], when training neural networks it is helpful to normalize the input data so that it is zero mean and so that more important dimensions of input data have a larger variance. We wanted a generic way to achieve this that would be invariant to arbitrary affine transforms of the input. The technique we developed requires as statistics a within-class covariance $\mathbf{W}$ and a between-class covariance $\mathbf{B}$, accumulated from the class-labeled data as if in preparation for multi-class Linear Discriminant Analysis (LDA). Assume in what follows that we have already normalized the data so that it is zero-mean. For the technique we are about to describe to make sense, the number of classes should not be much smaller than the feature dimension; fortunately, in our case it is much larger– $5000 > 300$, to give typical numbers.

Suppose we were to do multi-class LDA but not actually reduce the dimension. We would transform into a space where $\mathbf{W}$ was unit and $\mathbf{B}$ was diagonalized. Suppose the $\mathbf{B}$ in this space has diagonal elements b_i. Then the total covariance in each dimension i is $b_i + 1$. This has the desirable property that the data covariance is higher in "more important" directions, but it doesn't drop as fast as we'd like for unimportant directions– it never goes below 1. In our method, we do the LDA-type transform as mentioned above, then scale each row of the transform by $\sqrt{(b_i + 0.001)/(b_i + 1)}$. After this scaling, the total covariance becomes $b_i + 0.001$, where b_i is the ratio of between-class to within-class covariance. This seems to work well.

After creating the transform matrix as described above, we do a singular value decomposition on it, floor the singular values (normally to 5), and reconstruct again. The motivation here is to avoid a rarely encountered pathology that occurs when the training data covariance was close to singular, which leads to a transform with very large elements, that might produce very large transformed data values on mismatched test data or due to roundoff. This step rarely floors more than a handful of singular values so has little effect on the transform.

C.5 Parameter initialization

We decided not to implement generative pre-training as in [18], because while it is well established that it improves results for small datasets, our understanding is that as the amount of training data gets larger, it eventually gives no improvement compared to a suitable random initialization or discriminative layer-wise backpropagation as in [19]. We could not find a published reference for this; it is something we have been told verbally. We refer here specifically to speech recognition tasks; this does not apply to tasks like computer vision where much larger networks are used. In fact, the alternative "nnet1" implmentation of DNNs in Kaldi does support pre-training, and for small datasets (say, 50 hours or less), it generally gives slightly better results than the "nnet2" implementation which we speak of here. For larger datasets, the "nnet1" implementation eventually becomes impractical to run because it takes too long, and a detailed comparison is way beyond the scope of this paper.

Instead of pre-training, we use what is described in [19] as layer-wise back-propagation (BP). What this means is, we initialize a network with one hidden layer, train with BP for a short time (two "outer iterations" for our experiments reported here), then remove the final softmax layer and add a new, randomly initialized hidden layer on top of the existing hidden layer; train for a short time again; and repeat the process until we have the desired number of hidden layers. Similar to [20], we use a standard deviation of $\frac{1}{\sqrt{i}}$ for the weights, where i is the fan-in to the weight matrix; but we initialize the parameters of softmax layers to zero. Note: we found it essential to discard the parameters of the final softmax layer when adding each new hidden layer, as prescribed in [19].

For smaller datasets we can improve results versus layer-wise BP by initializing all but the last layer of the network from a network trained on another large dataset, possibly from another language. When initializing this way we typically find it best to use a larger network than we otherwise would have used.

Because we noticed that sometimes on an outer iteration immediately following the random initialization of parameters (including the first outer iteration), the parameter averaging can degrade rather than improve the objective function, we modified our parallel training method so that on these iterations, instead of averaging the parameters we choose the one that had the best objective function computed on the subset of data

that it was trained on (this choice of data avoids any extra computation).

C.6 Sequence training

Sequence training [21] is a term that has the the same meaning for DNNs that "discriminative training" [22] has in the speech recognition community for GMMs. It is a collective term for various objective functions used for training DNNs for sequence tasks, that only make sense at the whole-sequence level. This contrasts with the cross-entropy objective function which, given a fixed Viterbi alignment of the HMM states, easily decomposes over the frames of training data. In GMM-based speech recognition, the term "discriminative training" contrasts with Maximum Likelihood estimation; in DNN-based speech recognition it contrasts with cross-entropy training. There are two popular classes of sequence/discriminative objective functions:

- Maximum Mutual Information (MMI)-like objective functions [23, 22], more properly called conditional maximum likelihood: these have the form of sum over all utterances of the log-posterior of the correct word sequence for each utterance, given the model and the data. These include its popular 'boosted' variant [24] which is inspired by margin-based objective functions.
- Minimum Bayes Risk (MBR)-like objective functions: popular variants include Minimum Phone Error (MPE) [25, 22] and state-level Minimum Bayes Risk [26, 27]. These have the form of an expectation, given the data and the model, of an edit-distance type of error. We can compute its derivative w.r.t. the model parameters, because the posteriors of the different sequences vary with the model parameters.

This paper is mainly about our parallel approach to standard cross-entropy training. However, we also apply the same ideas (model averaging, preconditioning) to sequence training. We generally use state-level Minimum Bayes Risk (sMBR) [26, 27] although we have also implemented Minimum Phone Error (MPE) [25] and Boosted MMI [24]. The high-level details of our lattice-based training procedure are similar to [28], but note that in that paper we describe an alternative implementation of deep neural nets (the "nnet1" setup) that exists within Kaldi; this paper is about the alternative "nnet2" setup. Some items in common with the sequence training described in that paper include the following:

- We use a low, fixed learning rate (e.g. 0.00002).
- We generally train for about 4 epochs.

Some differences include the following:

- We do parallel SGD on multiple machines, with periodic model averaging.
- Rather than randomizing at the utterance level, we split the lattice into as small pieces as possible given the lattice topology, and excise parts of the lattice that would not contribute nonzero derivatives; and we randomize the order of the remaining pieces.
- To ensure that all layers of the network are trained about the same amount, we modify the learning rates in order to ensure that the relative change in parameters on each "outer iteration" is the same for each layer; their geometric average is constrained to equal the user-specified fixed learning rate (e.g. 0.00002) which we mentioned above.
- In our recipe, we generate the lattices only once.
- The minibatches actually consist of several small chunks of lattice (split as described above), from many different utterances, spliced together.

Something that we should note in connection with the learning rates is that for p-norm networks, since the network output is invariant to (nonzero) scaling of the parameters of the p-norm layers[2], and since the generalized weighted combination of Section C.2 may output arbitrarily scaled weights, it is hard to specify in advance a suitable learning rate. To solve this problem, we first scale the parameters of p-norm layers so so that the expected square of a randomly chosen matrix element is one.

For sequence training, because the frames in a minibatch are not drawn independently from the training data but consist of sequential frames from one or a few utterances, our "simple" preconditioning method is not applicable, and we only apply the online method.

C.7 Online decoding and iVector inputs

In speech recognition applications it is sometimes necessary to process data continuously as it arrives, so that there will be no latency in response. This makes it necessary that the algorithms used should not have any dependencies that are "backwards" in time. Backwards-in-time dependencies in our conventional neural net recipes, e.g. as reported in [14], include cepstral mean normalization (CMN), in which we subtract

[2]This is thanks to the "renormalization layers" that follow each p-norm layer [14]

the mean of the input features; and fMLLR adaptation, also known as constrained MLLR adaptation [29], in which we use a baseline GMM system to compute a likelihood-maximizing linear transform of the features. Although we use "online" versions of both of these things for online GMM-based decoding, it makes the system very complex and is not ideal for combination with DNNs.

In order to have a system that is easier to turn into an online algorithm, we use iVectors [30] as an additional input to the neural network, in addition to the spliced cepstral features. An iVector is a vector normally of dimension in the range of several hundred, that represents speaker characteristics in a form suitable for speaker identification, and which is extracted in a Maximum Likelihood way in conjunction with a single mixture-of-Gaussians model (their means are regressed on the iVector). The parameters of the factor analysis model that extracts the iVectors are trained without any supervision, just on a large number of audio recordings. In our case we extract iVectors of dimension 100. Once the iVector extractor is trained, we switch to "online" extraction of iVectors for both training and decoding, in which only frames preceding the current frame are taken as inputs to the iVector estimation process. At the beginning of the utterance, the iVector will be zero due to the prior term.

The actual inputs to the DNN in this setup normally consist of the iVector, plus ± 7 frames of Mel frequency cepstral coefficients (MFCCs) [31], without cepstral mean normalization. Some other authors [32] use log Mel filterbank energies; the MFCC features we use here are equivalent to log Mel filterbank energies because MFCCs are a linear transform of them (we use the same number of coefficients as filterbanks, 40 for these experiments) and our input data normalization (Section C.4) is invariant to such transforms; we only use MFCCs because they are more easily compressible and our "training example" data structure (Section 3.3) compresses the input features.

In order to train models that are well matched both to per-speaker decoding, where statistics from previous utterances of the same speaker are included in the iVector estimation, and per-utterance decoding, where we make a fresh start each time, we generally train after splitting the speakers into "fake" speakers that each have no more than two utterances.

In experiments on a number of datasets, we have generally found that this method gives us about the same performance as our previous recipe where we trained a DNN on top of ± 4 frames of the standard 40-dimensional features consisting of mean-normalized MFCC features processed with LDA and MLLT, and speaker adapted with fMLLR (a.k.a. constrained MLLR [29]). We prefer it due to its convenience for applications and its convenience for cross-system transfer learning.