Training Neural Networks with Stochastic Hessian-Free Optimization

Ryan Kiros
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
rkiros@ualberta.ca

Abstract

Hessian-free (HF) optimization has been successfully used for training deep autoencoders and recurrent networks. HF uses the conjugate gradient algorithm to construct update directions through curvature-vector products that can be computed on the same order of time as gradients. In this paper we exploit this property and study stochastic HF with gradient and curvature mini-batches independent of the dataset size. We modify Martens' HF for these settings and integrate dropout, a method for preventing co-adaptation of feature detectors, to guard against overfitting. Stochastic Hessian-free optimization gives an intermediary between SGD and HF that achieves competitive performance on both classification and deep autoencoder experiments.

1 Introduction

Stochastic gradient descent (SGD) has become the most popular algorithm for training neural networks. Not only is SGD simple to implement but its noisy updates often leads to solutions that are well-adapt to generalization on held-out data [1]. Furthermore, SGD operates on small mini-batches potentially allowing for scalable training on large datasets. For training deep networks, SGD can be used for fine-tuning after layerwise pre-training [2] which overcomes many of the difficulties of training deep networks. Additionally, SGD can be augmented with dropout [3] as a means of preventing overfitting.

There has been recent interest in second-order methods for training deep networks, partially due to the successful adaptation of Hessian-free (HF) by [4], an instance of the more general family of truncated Newton methods. Second-order methods operate in batch settings with less but more substantial weight updates. Furthermore, computing gradients and curvature information on large batches can easily be distributed across several machines. Martens' HF was able to successfully train deep autoencoders without the use of pre-training and was later used for solving several pathological tasks in recurrent networks [5].

HF iteratively proposes update directions using the conjugate gradient algorithm, requiring only curvature-vector products and not an explicit computation of the curvature matrix. Curvature-vector products can be computed on the same order of time as it takes to compute gradients with an additional forward and backward pass through the function's computational graph [6, 7]. In this paper we exploit this property and introduce stochastic Hessian-free optimization (SHF), a variation of HF that operates on gradient and curvature mini-batches independent of the dataset size. Our goal in developing SHF is to combine the generalization advantages of SGD with second-order information from HF. SHF can adapt its behaviour through the choice of batch size and number of conjugate gradient iterations, for which its behaviour either becomes more characteristic of SGD or HF. Additionally we integrate dropout, as a means of preventing co-adaptation of feature detectors. We

perform experimental evaluation on both classification and deep autoencoder tasks. For classification, dropout SHF is competitive with dropout SGD on all tasks considered while for autoencoders SHF performs comparably to HF and momentum-based methods. Moreover, no tuning of learning rates needs to be done.

2 Related work

Much research has been investigated into developing adaptive learning rates or incorporating second-order information into SGD. [8] proposed augmenting SGD with a diagonal approximation of the Hessian while Adagrad [9] uses a global learning rate while dividing by the norm of previous gradients in its update. SGD with Adagrad was shown to be beneficial in training deep distributed networks for speech and object recognition [10]. To completely avoid tuning learning rates, [11] considered computing rates as to minimize estimates of the expectation of the loss at any one time. [12] proposed SGD-QN for incorporating a quasi-Newton approximation to the Hessian into SGD and used this to win one of the 2008 PASCAL large scale learning challenge tracks. Recently, [13] provided a relationship between HF, Krylov subspace descent and natural gradient due to their use of the Gauss-Newton curvature matrix. Furthermore, [13] argue that natural gradient is robust to overfitting as well as the order of the training samples. Other methods incorporating the natural gradient such as TONGA [14] have also showed promise on speeding up neural network training.

Analyzing the difficulty of training deep networks was done by [15], proposing a weight initialization that demonstrates faster convergence. More recently, [16] argue that large neural networks waste capacity in the sense that adding additional units fail to reduce underfitting on large datasets. The authors hypothesize the SGD is the culprit and suggest exploration with stochastic natural gradient or stochastic second-order methods. Such results further motivate our development of SHF. [17] show that with careful attention to the parameter initialization and momentum schedule, first-order methods can be competitive with HF for training deep autoencoders and recurrent networks. We compare against these methods in our autoencoder evaluation.

Related to our work is that of [18], who proposes a dynamic adjustment of gradient and curvature mini-batches for HF with convex losses based on variance estimations. Unlike our work, the batch sizes used are dynamic with a fixed ratio and are initialized as a function of the dataset size. Other work on using second-order methods for neural networks include [19] who proposed using the Jacobi pre-conditioner for HF, [20] using HF to generate text in recurrent networks and [21] who explored training with Krylov subspace descent (KSD). Unlike HF, KSD could be used with Hessian-vector products but requires additional memory to store a basis for the Krylov subspace. L-BFGS has also been successfully used in fine-tuning pre-trained deep autoencoders, convolutional networks [22] and training deep distributed networks [10]. Other developments and detailed discussion of gradient-based methods for neural networks is described in [23].

3 Hessian-free optimization

In this section we review Hessian-free optimization, largely following the implementation of Martens [4]. We refer the reader to [24] for detailed development and tips for using HF.

We consider unconstrained minimization of a function $f : \mathbb{R}^n \to \mathbb{R}$ with respect to parameters θ. More specifically, we assume f can be written as a composition $f(\theta) = L(F(\theta))$ where L is a convex loss function and $F(\theta)$ is the output of a neural network with ℓ non-input layers. We will mostly focus on the case when f is non-convex. Typically L is chosen to be a matching loss to a corresponding transfer function $p(z) = p(F(\theta))$. For a single input, the $(i+1)$-th layer of the network is expressed as

$$y_{i+1} = s_i(W_i\, y_i + b_i) \tag{1}$$

where s_i is a transfer function, W_i is the weights connecting layers i and $i+1$ and b_i is a bias vector. Common transfer functions include the sigmoid $s_i(x) = (1 + \exp(-x))^{-1}$, the hyperbolic tangent $s_i(x) = \tanh(x)$ and rectified linear units $s_i(x) = \max(x, 0)$. In the case of classification tasks, the

loss function used is the generalized cross entropy and softmax transfer

$$L(p(z), t) = -\sum_{j=1}^{k} t_j \log(p(z_j)), \quad p(z_j) = \exp(z_j) / \sum_{l=1}^{k} \exp(z_l) \tag{2}$$

where k is the number of classes, t is a target vector and z_j the j-th component of output vector z. Consider a local quadratic approximation $M_\theta(\delta)$ of f around θ:

$$f(\theta + \delta) \approx M_\theta(\delta) = f(\theta) + \nabla f(\theta)^T \delta + \frac{1}{2} \delta^T B \delta \tag{3}$$

where $\nabla f(\theta)$ is the gradient of f and B is the Hessian or an approximation to the Hessian. If f was convex, then $B \succeq 0$ and equation 3 exhibits a minimum δ^*. In Newton's method, θ_{k+1}, the parameters at iteration $k+1$, are updated as $\theta_{k+1} = \theta_k + \alpha_k \delta_k^*$ where $\alpha_k \in [0, 1]$ is the rate and δ_k^* is computed as

$$\delta_k^* = -B^{-1} \nabla f(\theta_{k-1}) \tag{4}$$

for which calculation requires $O(n^3)$ time and thus often prohibitive. Hessian-free optimization alleviates this by using the conjugate gradient (CG) algorithm to compute an approximate minimizer δ_k. Specifically, CG minimizes the quadratic objective $q(\delta)$ given by

$$q(\delta) = \frac{1}{2} \delta^T B \delta + \nabla f(\theta_{k-1})^T \delta \tag{5}$$

for which the corresponding minimizer of $q(\delta)$ is $-B^{-1} \nabla f(\theta_{k-1})$. The motivation for using CG is as follows: while computing B is expensive, compute the product Bv for some vector v can be computed on the same order of time as it takes to compute $\nabla f(\theta_{k-1})$ using the R-operator [6]. Thus CG can efficiently compute an iterative solution to the linear system $B\delta_k = -\nabla(f(\theta_{k-1}))$ corresponding to a new update direction δ_k.

When f is non-convex, the Hessian may not be positive semi-definite and thus equation 3 no longer has a well defined minimum. Following Martens, we instead use the generalized Gauss-newton matrix defined as $B = J^T L'' J$ where J is the Jacobian of f and L'' is the Hessian of L [1]. So long as $f(\theta) = L(F(\theta))$ for convex L then $B \succeq 0$. Given a vector v, the product $Bv = J^T L'' J v$ is computed successively by first computing Jv, then $L''(Jv)$ and finally $J^T(L'' Jv)$ [7]. To compute Jv, we utilize the R-operator. The R-operator of $F(\theta)$ with respect to v is defined as

$$\mathcal{R}_v\{F(\theta)\} = \lim_{\epsilon \to 0} \frac{F(\theta + \epsilon v) - F(\theta)}{\epsilon} = Jv \tag{6}$$

Computing $\mathcal{R}_v\{F(\theta)\}$ in a neural network is easily done using a forward pass by computing $\mathcal{R}_v\{y_i\}$ for each layer output y_i. More specifically,

$$\mathcal{R}_v\{y_{i+1}\} = \mathcal{R}_v\{W_i\, y_i + b_i\} s_i' = (v(W_i) y_i \; + v(b_i) + W_i \mathcal{R}\{y_i\}) s_i' \tag{7}$$

where $v(W_i)$ is the components of v corresponding to parameters between layers i and $i+1$ and $\mathcal{R}\{y_1\} = 0$ (where y_1 is the input data). In order to compute $J^T(L'' Jv)$, we simply apply back-propagation but using the vector $L'' Jv$ instead of ∇L as is usually done to compute ∇f. Thus, Bv may be computed through a forward and backward pass in the same sense that L and $\nabla f = J^T \nabla L$ are.

As opposed to minimizing equation 3, Martens instead uses an additional damping parameter λ with damped quadratic approximation

$$\hat{M}_\theta(\delta) = f(\theta) + \nabla f(\theta)^T \delta + \frac{1}{2} \delta^T \hat{B} \delta = f(\theta) + \nabla f(\theta)^T \delta + \frac{1}{2} \delta^T (B + \lambda I) \delta \tag{8}$$

Damping the quadratic through λ gives a measure of how conservative the quadratic approximation is. A large value of λ is more conservative and as $\lambda \to \infty$ updates become similar to stochastic gradient descent. Alternatively, a small λ allows for more substantial parameter updates especially

[1] While an abuse of definition, we still refer to "curvature-vector products" and "curvature batches" even when B is used.

along low curvature directions. Martens dynamically adjusts λ at each iteration using a Levenberg-Marquardt style update based on computing the reduction ratio

$$\rho = (f(\theta + \delta) - f(\theta))/(M_\theta(\delta) - M_\theta(0)) \tag{9}$$

If ρ is sufficiently small or negative, λ is increased while if ρ is large then λ is decreased. The number of CG iterations used to compute δ has a dramatic effect on ρ which is further discussed in section 4.1.

To accelerate CG, Martens makes use of the diagonal pre-conditioner

$$P = \left[\text{diag}\left(\sum_{j=1}^{m} \nabla f^{(j)}(\theta) \odot \nabla f^{(j)}(\theta)\right) + \lambda I\right]^{\xi} \tag{10}$$

where $f^{(j)}(\theta)$ is the value of f for datapoint j and $\odot$ denotes component-wise multiplication. P can be easily computed on the same backward pass as computing ∇f.

Finally, two backtracking methods are used: one after optimizing CG to select δ and the other a backtracking linesearch to compute the rate α. Both these methods operate in the standard way, backtracking through proposals until the objective no longer decreases.

4 Stochastic Hessian-free optimization

Martens' implementation utilizes the full dataset for computing objective values and gradients, and mini-batches for computing curvature-vector products. Naively setting both batch sizes to be small causes several problems. In this section we describe these problems and our contributions in modifying Martens' original algorithm to this setting.

4.1 Short CG runs, δ-momentum and use of mini-batches

The CG termination criteria used by Martens is based on a measure of relative progress in optimizing $\hat{M}_\theta$. Specifically, if x_j is the solution at CG iteration j, then training is terminated when

$$\frac{\hat{M}_\theta(x_j) - \hat{M}_\theta(x_{j-k})}{\hat{M}_\theta(x_j)} < \epsilon \tag{11}$$

where k =max$(10, j/10)$ and ϵ is a small positive constant. The effect of this stopping criteria has a dependency on the strength of the damping parameter λ, among other attributes such as the current parameter settings. For sufficiently large λ, CG only requires 10-20 iterations when a pre-conditioner is used. As λ decreases, more iterations are required to account for pathological curvature that can occur in optimizing f and thus leads to more expensive CG iterations. Such behavior would be undesirable in a stochastic setting where preference would be put towards having equal length CG iterations throughout training. To account for this, we fix the number of CG iterations to be only 3-5 across training for classification and 25-50 for training deep autoencoders. Let ζ denote this cut-off. Setting a limit on the number of CG iterations is used by [4] and [20] and also has a damping effect, since the objective function and quadratic approximation will tend to diverge as CG iterations increase [24]. We note that due to the shorter number of CG runs, the iterates from each solution are used during the CG backtracking step.

A contributor to the success of Martens' HF is the use of information sharing across iterations. At iteration k, CG is initialized to be the previous solution of CG from iteration $k-1$, with a small decay. For the rest of this work, we denote this as δ-momentum. δ-momentum helps correct proposed update directions when the quadratic approximation varies across iterations, in the same sense that momentum is used to share gradients. This momentum interpretation was first suggested by [24] in the context of adapting HF to a setting with short CG runs. Unfortunately, the use of δ-momentum becomes challenging when short CG runs are used. Given a non-zero CG initialization, $\hat{M}_\theta$ may be more likely to remain positive after terminating CG and assuming $f(\theta + \delta) - f(\theta) < 0$, means that the reduction ratio will be negative and thus λ will be increased to compensate. While this is not necessarily unwanted behavior, having this occur too frequently will push SHF to be too conservative and possibly result in the backtracking linesearch to reject proposed updates. Our

solution is to utilize a schedule on the amount of decay used on the CG starting solution. This is motivated by [24] suggesting more attention on the CG decay in the setting of using short CG runs. Specifically, if δ_k^0 is the initial solution to CG at iteration k, then

$$\delta_k^0 = \gamma_e \delta_{k-1}^{\zeta}, \quad \gamma_e = \min(1.01\gamma_{e-1}, .99) \tag{12}$$

where γ_e is the decay at epoch e, $\delta_1^0 = 0$ and $\gamma_1 = 0.5$. While in batch training a fixed γ is suitable, in a stochastic setting it is unlikely that a global decay parameter is sufficient. Our schedule has an annealing effect in the sense that γ values near 1 are feasible late in training even with only 3-5 CG iterations, a property that is otherwise hard to achieve. This allows us to benefit from sharing more information across iterations late in training, similar to that of a typical momentum method.

A remaining question to consider is how to set the sizes of the gradient and curvature mini-batches. [24] discuss theoretical advantages to utilizing the same mini-batches for computing the gradient and curvature vector products. In our setting, this may lead to some difficulties. Using same-sized batches allows $\lambda \to 0$ during training [24]. Unfortunately, this can become incompatible with our short hard-limit on the number of CG iterations, since CG requires more work to optimize $\hat{M}_\theta$ when λ approaches zero. To account for this, on classification tasks where 3-5 CG iterations are used, we opt to use gradient mini-batches that are 5-10 times larger than curvature mini-batches. For deep autoencoder tasks where more CG iterations are used, we instead set both gradient and curvature batches to be the same size. The behavior of λ is dependent on whether or not dropout is used during training. Figure 1 demonstrates the behavior of λ during classification training with and without the use of dropout. With dropout, λ no longer converges to 0 but instead plummets, rises and flattens out. In both settings, λ does not decrease substantially as to negatively effect the proposed CG solution and consequently the reduction ratio. Thus, the amount of work required by CG remains consistent late in training. The other benefit to using larger gradient batches is to account for the additional computation in computing curvature-vector products which would make training longer if both mini-batches were small and of the same size. In [4], the gradients and objectives are computed using the full training set throughout the algorithm, including during CG backtracking and the backtracking linesearch. We utilize the gradient mini-batch for the current iteration in order to compute all necessary gradient and objectives throughout the algorithm.

4.2 Levenberg-Marquardt damping

Martens makes use of the following Levenberg-Marquardt style damping criteria for updating λ:

$$\textbf{if} \rho > \frac{3}{4}, \lambda \leftarrow \frac{2}{3}\lambda \ \textbf{elseif} \rho < \frac{1}{4}, \lambda \leftarrow \frac{3}{2}\lambda \tag{13}$$

which given a suitable initial value will converge to zero as training progresses. We observed that the above damping criteria is too harsh in the stochastic setting in the sense that λ will frequently oscillate, which is sensible given the size of the curvature mini-batches. We instead opt for a much softer criterion, for which lambda is updated as

$$\textbf{if} \rho > \frac{3}{4}, \lambda \leftarrow \frac{99}{100}\lambda \ \textbf{elseif} \rho < \frac{1}{4}, \lambda \leftarrow \frac{100}{99}\lambda \tag{14}$$

This choice, although somewhat arbitrary, is consistently effective. Thus reduction ratio values computed from curvature mini-batches will have less overall influence on the damping strength.

4.3 Integrating dropout

Dropout is a recently proposed method for improving the training of neural networks. During training, each hidden unit is omitted with a probability of 0.5 along with optionally omitting input features similar to that of a denoising autoencoder [25]. Dropout can be viewed in two ways. By randomly omitting feature detectors, dropout prevents co-adaptation among detectors which can improve generalization accuracy on held-out data. Secondly, dropout can be seen as a type of model averaging. At test time, outgoing weights are halved. If we consider a network with a single hidden layer and k feature detectors, using the mean network at test time corresponds to taking the geometric average of 2^k networks with shared weights. Dropout is integrated in stochastic HF by randomly omitting feature detectors on both gradient and curvature mini-batches from the last hidden layer

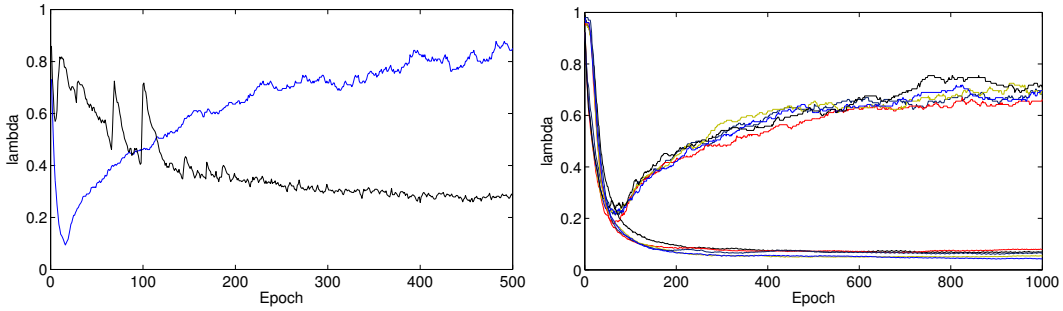

Figure 1: Values of the damping strength λ during training of MNIST (left) and USPS (right) with and without dropout using $\lambda = 1$ for classification. When dropout is included, the damping strength initially decreases followed by a steady increase over time.

during each iteration. Since we assume that the curvature mini-batches are a subset of the gradient mini-batches, the same feature detectors are omitted in both cases.

Since the curvature estimates are noisy, it is important to consider the stability of updates when different stochastic networks are used in each computation. The weight updates in dropout SGD are augmented with momentum not only for stability but also to speed up learning. Specifically, at iteration k the parameter update is given by

$$\Delta\theta_k = p_k \Delta\theta_{k-1} - (1 - p_k)\alpha_k \langle \nabla f \rangle, \quad \theta_k = \theta_{k-1} + \Delta\theta_k \tag{15}$$

where p_k and a_k are the momentum and learning rate, respectively. We incorporate an additional exponential decay term β_e when performing parameter updates. Specifically, each parameter update is computed as

$$\theta_k = \theta_{k-1} + \beta_e \alpha_k \delta_k, \quad \beta_e = c\beta_{e-1} \tag{16}$$

where $c \in (0, 1]$ is a fixed parameter chosen by the user. Incorporating β_e into the updates, along with the use of δ-momentum, leads to more stable updates and fine convergence particularly when dropout is integrated during training.

4.4 Algorithm

Pseudo-code for one iteration of our implementation of stochastic Hessian-free is presented. Given a gradient minibatch X_k^g and curvature minibatch X_k^c, we first sample dropout units (if applicable) for the inputs and last hidden layer of the network. These take the form of a binary vector, which are multiplied component-wise by the activations y_i. In our pseudo-code, $\text{CG}(\delta_k^0, \nabla f, P, \zeta)$ is used to denote applying CG with initial solution δ_k^0, gradient ∇f, pre-conditioner P and ζ iterations. Note that, when computing δ-momentum, the ζ-th solution in iteration $k - 1$ is used as opposed to the solution chosen via backtracking. Given the objectives f_{k-1} computed with θ and f_k computed with $\theta + \delta_k$, the reduction ratio ρ is calculated utilizing the un-damped quadratic approximation $M_\theta(\delta_k)$. This allows updating λ using the Levenberg-Marquardt style damping. Finally, a backtracking line-search with at most ω steps is performed to compute the rate and serves as a last defense against potentially poor update directions.

Since curvature mini-batches are sampled from a subset of the gradient mini-batch, it is then sensible to utilize different curvature mini-batches on different epochs. Along with cycling through gradient mini-batches during each epoch, we also cycle through curvature subsets every h epochs, where h is the size of the gradient mini-batches divided by the size of the curvature mini-batches. For example, if the gradient batch size is 1000 and the curvature batch size is 100, then curvature mini-batch sampling completes a full cycle every 1000/100 = 10 epochs.

Finally, one simple way to speed up training as indicated in [24], is to cache the activations when initially computing the objective f_k. While each iteration of CG requires computing a curvature-vector product, the network parameters are fixed during CG and is thus wasteful to re-compute the network activations on each iteration.

Algorithm 1 Stochastic Hessian-Free Optimization

$X_k^g \leftarrow$ gradient minibatch, $X_k^c \leftarrow$ curvature minibatch, $|X_k^g| = h|X_k^c|, h \in \mathbb{Z}^+$
Sample dropout units for inputs and last hidden layer
if start of new epoch **then**
  $\gamma_e \leftarrow \min(1.01\gamma_{e-1}, .99)$ {δ-momentum}
end if
$\delta_k^0 \leftarrow \gamma_e \delta_{k-1}^{\zeta}$
$f_{k-1} \leftarrow f(X_k^g; \theta), \nabla f \leftarrow \nabla f(X_k^g; \theta), P \leftarrow \text{Precon}(X_k^g; \theta)$
Solve $(B + \lambda I)\delta_k = -\nabla f$ using $\text{CG}(\delta_k^0, \nabla f, P, \zeta)$ {Using X_k^c to compute $B\delta_k$}
$f_k \leftarrow f(X_k^g; \theta + \delta_k)$ {CG backtracking}
for $j = \zeta$ - 1 to 1 **do**
  $f(\theta + \delta_k^j) \leftarrow f(X_k^g; \theta + \delta_k^j)$
  if $f(\theta + \delta_k^j) < f_k$ **then**
    $f_k \leftarrow f(\theta + \delta_k^j), \delta_k \leftarrow \delta_k^j$
  end if
end for
$\rho \leftarrow (f_k - f_{k-1})/(\frac{1}{2}\delta_k^T B \delta_k + \nabla f^T \delta_k)$ {Using X_k^c to compute $B\delta_k$}
if $\rho < .25$, $\lambda \leftarrow 1.01\lambda$ **elseif** $\rho > .75$, $\lambda \leftarrow .99\lambda$ **end if**
$\alpha_k \leftarrow 1, j \leftarrow 0$ {Backtracking linesearch}
while $j < \omega$ **do**
  if $f_k > f_{k-1} + .01\alpha_k \nabla f^T \delta_k$ **then** $\alpha_k \leftarrow .8\alpha_k, j \leftarrow j + 1$ **else break end if**
end while
$\theta \leftarrow \theta + \beta_e \alpha_k \delta_k, k \leftarrow k + 1$ {Parameter update}

5 Experiments

We perform experimental evaluation on both classification and deep autoencoder tasks. The goal of classification experiments is to determine the effectiveness of SHF on test error generalization. For autoencoder tasks, we instead focus just on measuring the effectiveness of the optimizer on the training data. The datasets and experiments are summarized as follows:

- MNIST: Handwritten digits of size 28×28 with 60K training samples and 10K testing samples. For classification, we train networks of size 784-1200-1200-10 with rectifier activations. For deep autoencoders, the encoder architecture of 784-1000-500-250-30 with a symmetric decoding architecture is used. Logistic activations are used with a binary cross entropy error. For classification experiments, the data is scaled to have zero mean and unit variance.
- CURVES: Artificial dataset of curves of size 28×28 with 20K training samples and 10K testing samples. We train a deep autoencoder using an encoding architecture of 784-400-200-100-50-25-6 with symmetric decoding. Similar to MNIST, logistic activations and binary cross entropy error are used.
- USPS: Handwritten digits of size 16×16 with 11K examples. We perform classification using 5 randomly sampled batches of 8K training examples and 3K testing examples as in [26] Each batch has an equal number of each digit. Classification networks of size 256-500-500-10 are trained with rectifier activations. The data is scaled to have zero mean and unit variance.
- Reuters: A collection of 8293 text documents from 65 categories. Each document is represented as a 18900-dimensional bag-of-words vector. Word counts C are transformed to $\log(1 + C)$ as is done by [3]. The publically available train/test split of is used. We train networks of size 18900-65 for classification due to the high dimensionality of the inputs, which reduces to softmax-regression.

For classification experiments, we perform comparison of SHF with and without dropout against dropout SGD [3]. All classification experiments utilize the sparse initialization of Martens [4] with initial biases set to 0.1. The sparse initialization in combination with ReLUs make our networks similar to the deep sparse rectifier networks of [28]. All algorithms are trained for 500 epochs on MNIST and 1000 epochs on USPS and Reuters. We use weight decay of 5×10^{-4} for SHF and 2×10^{-5} for dropout SHF. A held-out validation set was used for determining the amount of input

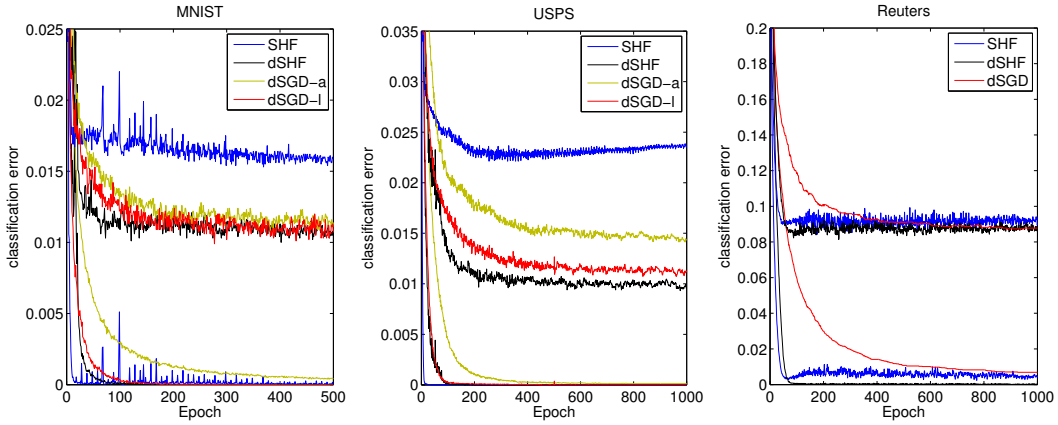

Figure 2: Training and testing curves for classification. dSHF: dropout SHF, dSGD: dropout SGD, dSGD-a: dropout on all layers, dSGD-l: dropout on last hidden layer only (as well as the inputs).

dropout for all algorithms. Both SHF and dropout SHF use initial damping of $\lambda = 1$, gradient batch size of 1000, curvature batch size of 100 and 3 CG iterations per batch.

Dropout SGD training uses an exponential decreasing learning rate schedule initialized at 10, in combination with max-norm weight clipping [3]. This allows SGD to use larger learning rates for greater exploration early in training. A linearly increasing momentum schedule is used with initial momentum of 0.5 and final momentum of 0.99. No weight decay is used. For additional comparison we also train dropout SGD when dropout is only used in the last hidden layer, as is the case with dropout SHF.

For deep autoencoder experiments, we use the same experimental setup as in Chapter 7 of [17]. In particular, we focus solely on training error without any L2 penalty in order to determine the effectiveness of the optimizer on modeling the training data. Comparison is made against SGD, SGD with momentum, HF and Nesterov's accelerated gradient (NAG). On CURVES, SHF uses an initial damping of $\lambda = 10$, gradient and curvature batch sizes of 2000 and 25 CG iterations per batch. On MNIST, we use initial $\lambda = 1$, gradient and curvature batch sizes of 3000 and 50 CG iterations per batch. Autoencoder training is ran until no sufficient progress is made, which occurs at around 250 epochs on CURVES and 100 epochs on MNIST.

5.1 Classification results

Figure 2 summarizes our classification results. At epoch 500, dropout SHF achieves 107 errors on MNIST. This result is similar to [3] which achieve 100-115 errors with various network sizes when training for a few thousand epochs. Without dropout or input corruption, SHF achieves 159 errors on MNIST, on par with existing methods that do not incorporate prior knowledge, pre-training, image distortions or dropout. As with [4], we hypothesize that further improvements can be made by fine-tuning with SHF after unsupervised layerwise pre-training.

After 1000 epochs of training on five random splits of USPS, we obtain final classification errors of 1%, 1.1%, 0.8%, 0.9% and 0.97% with a mean test error of 0.95%. Both algorithms use 50% input corruption. For additional comparison, [29] obtains a mean classification error of 1.14% using a pre-trained deep network for large-margin nearest neighbor classification with the same size splits. Without dropout, SHF overfits the training data.

On the Reuters dataset, SHF with and without dropout both demonstrate accelerated training. We hypothesize that further speedup may also be obtained by starting training with a much smaller λ initialization, which we suspect is conservative given that the problem is convex.

Table 1: Training errors on the deep autoencoder tasks. All results are obtained from [17]. M(0.99) refers to momentum capped at 0.99 and similarily for M(0.9). SGD-VI refers to SGD using the variance normalized initialization of [15].

problem	NAG	M(0.99)	M(0.9)	SGD	SGD-VI [19]	HF	SHF
CURVES	0.078	0.110	0.220	0.250	0.160	0.110	0.089
MNIST	0.730	0.770	0.990	1.100	0.900	0.780	0.877

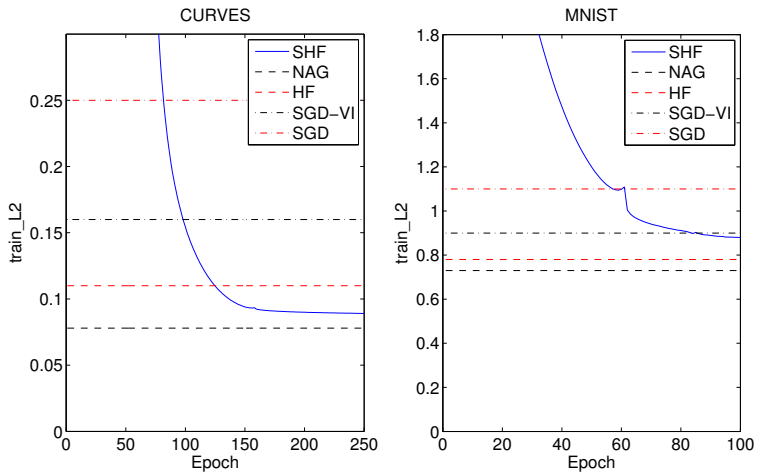

Figure 3: Learning curves for the deep autoencoder tasks. The CG decay parameter γ is shut off at epoch 160 on CURVES and epoch 60 on MNIST.

5.2 Deep autoencoder results

Figure 3 and table 1 summarize our results. Inspired by [17] we make one additional modification to our algorithms. As soon as training begins to diverge, we turn off the CG decay parameter γ in a similar fashion as the the momentum parameter μ is decreased in [17]. When $\gamma = 0$, CG is no longer initialized from the previous solution and is instead initialized to zero. As with [17], this has a dramatic effect on the training error but to a lesser extent as momentum and Nesterov's accelerated gradient. [17] describes the behaviour of this effect as follows: with a large momentum, the optimizer is able to make steady progress along slow changing directions of low curvature. By decreasing the momentum late in training, the optimizer is then able to quickly reach a local minimum from finer optimization along high curvature directions, which would otherwise be too difficult to obtain with an aggressive momentum schedule. This observation further motivates the relationship between momentum and information sharing through CG.

Our experimental results demonstrate that SHF does not perform significantly better or worse on these datasets compared to existing approaches. It is able to outperform HF on CURVES but not on MNIST. An attractive property that is shared with both HF and SHF is not requiring the careful schedule tuning that is necessary for momentum and NAG. We also attempted experiments with SHF using the same setup for classification with smaller batches and 5 CG iterations. The results were worse: on CURVES the lowest training error obtained was 0.19. This shows that while such a setup is useful from the viewpoint of noisy updates and test generalization, they hamper the effectiveness of making progress on hard to optimize regions.

6 Conclusion

In this paper we proposed a stochastic variation of Martens' Hessian-free optimization incorporating dropout for training neural networks on classification and deep autoencoder tasks. By adapting the batch sizes and number of CG iterations, SHF can be constructed to perform well for classification

against dropout SGD or optimizing deep autoencoders comparing HF, NAG and momentum methods. While our initial results are promising, of interest would be adapting stochastic Hessian-free optimization to other network architectures:

- **Convolutional networks.** The most common approach to training convolutional networks has been SGD incorporating a diagonal Hessian approximation [8]. Dropout SGD was recently used for training a deep convolutional network on ImageNet [30].
- **Recurrent Networks.** It was largely believed that RNNs were too difficult to train with SGD due to the exploding/vanishing gradient problem. In recent years, recurrent networks have become popular again due to several advancements made in their training [31].
- **Recursive Networks.** Recursive networks have been successfully used for tasks such as sentiment classification and compositional modeling of natural language from word embeddings [32]. These architectures are usually trained using L-BFGS.

It is not clear yet whether this setup is easily generalizable to the above architectures or whether improvements need to be considered. Furthermore, additional experimental comparison would involve dropout SGD with the adaptive methods of Adagrad [9] or [11], as well as the importance of pre-conditioning CG. None the less, we hope that this work initiates future research in developing stochastic Hessian-free algorithms.

Acknowledgments

The author would like to thank Csaba Szepesvári for helpful discussion as well as David Sussillo for his guidance when first learning about and implementing HF. The author would also like to thank the anonymous ICLR reviewers for their comments and suggestions.

References

[1] L. Bottou and O. Bousquet. The tradeoffs of large-scale learning. *Optimization for Machine Learning*, page 351, 2011.

[2] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. *NIPS*, 19:153, 2007.

[3] G.E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R.R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580*, 2012.

[4] J. Martens. Deep learning via hessian-free optimization. In *ICML*, volume 951, 2010.

[5] J. Martens and I. Sutskever. Learning recurrent neural networks with hessian-free optimization. In *ICML*, 2011.

[6] B.A. Pearlmutter. Fast exact multiplication by the hessian. *Neural Computation*, 6(1):147–160, 1994.

[7] N.N. Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. *Neural computation*, 14(7):1723–1738, 2002.

[8] Y. LeCun, L. Bottou, G. Orr, and K. Müller. Efficient backprop. *Neural networks: Tricks of the trade*, pages 546–546, 1998.

[9] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*, 12:2121–2159, 2010.

[10] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, A. Senior, P. Tucker, K. Yang, et al. Large scale distributed deep networks. In *NIPS*, pages 1232–1240, 2012.

[11] T. Schaul, S. Zhang, and Y. LeCun. No more pesky learning rates. *arXiv:1206.1106*, 2012.

[12] A. Bordes, L. Bottou, and P. Gallinari. Sgd-qn: Careful quasi-newton stochastic gradient descent. *JMLR*, 10:1737–1754.

[13] Razvan Pascanu and Yoshua Bengio. Natural gradient revisited. *arXiv preprint arXiv:1301.3584*, 2013.

[14] N. Le Roux, P.A. Manzagol, and Y. Bengio. Topmoumoute online natural gradient algorithm. In *NIPS*, 2007.

[15] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010.

[16] Yann N Dauphin and Yoshua Bengio. Big neural networks waste capacity. *arXiv preprint arXiv:1301.3583*, 2013.

[17] I. Sutskever. *Training Recurrent Neural Networks*. PhD thesis, University of Toronto, 2013.

[18] R.H. Byrd, G.M. Chin, J. Nocedal, and Y. Wu. Sample size selection in optimization methods for machine learning. *Mathematical Programming*, pages 1–29, 2012.

[19] O. Chapelle and D. Erhan. Improved preconditioner for hessian free optimization. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[20] I. Sutskever, J. Martens, and G. Hinton. Generating text with recurrent neural networks. In *ICML*, 2011.

[21] O. Vinyals and D. Povey. Krylov subspace descent for deep learning. *arXiv:1111.4259*, 2011.

[22] Q.V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A.Y. Ng. On optimization methods for deep learning. In *ICML*, 2011.

[23] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. *arXiv:1206.5533*, 2012.

[24] J. Martens and I. Sutskever. Training deep and recurrent networks with hessian-free optimization. *Neural Networks: Tricks of the Trade*, pages 479–535, 2012.

[25] P. Vincent, H. Larochelle, Y. Bengio, and P.A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *ICML*, pages 1096–1103, 2008.

[26] R. Min, D.A. Stanley, Z. Yuan, A. Bonner, and Z. Zhang. A deep non-linear feature mapping for large-margin knn classification. In *Ninth IEEE International Conference on Data Mining*, pages 357–366. IEEE, 2009.

[27] Deng Cai, Xuanhui Wang, and Xiaofei He. Probabilistic dyadic data analysis with local and global consistency. In *ICML*, pages 105–112, 2009.

[28] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *AISTATS*, 2011.

[29] R. Min, D.A. Stanley, Z. Yuan, A. Bonner, and Z. Zhang. A deep non-linear feature mapping for large-margin knn classification. In *ICDM*, pages 357–366, 2009.

[30] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. *NIPS*, 25, 2012.

[31] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu. Advances in optimizing recurrent networks. *arXiv:1212.0901*, 2012.

[32] R. Socher, B. Huval, C.D. Manning, and A.Y. Ng. Semantic compositionality through recursive matrix-vector spaces. In *EMNLP*, pages 1201–1211, 2012.