Deep learning via Hessian-free optimization

James Martens
University of Toronto, Ontario, M5S 1A1, Canada

JMARTENS@CS.TORONTO.EDU

Abstract

We develop a 2nd-order optimization method based on the "Hessian-free" approach, and apply it to training deep auto-encoders. Without using pre-training, we obtain results superior to those reported by Hinton & Salakhutdinov (2006) on the same tasks they considered. Our method is practical, easy to use, scales nicely to very large datasets, and isn't limited in applicability to auto-encoders, or any specific model class. We also discuss the issue of "pathological curvature" as a possible explanation for the difficulty of deep-learning and how 2nd-order optimization, and our method in particular, effectively deals with it.

1. Introduction

Learning the parameters of neural networks is perhaps one of the most well studied problems within the field of machine learning. Early work on backpropagation algorithms showed that the gradient of the neural net learning objective could be computed efficiently and used within a gradient-descent scheme to learn the weights of a network with multiple layers of non-linear hidden units. Unfortunately, this technique doesn't seem to generalize well to networks that have very many hidden layers (i.e. deep networks). The common experience is that gradient-descent progresses extremely slowly on deep nets, seeming to halt altogether before making significant progress, resulting in poor performance on the training set (under-fitting).

It is well known within the optimization community that gradient descent is unsuitable for optimizing objectives that exhibit pathological curvature. 2nd-order optimization methods, which model the local curvature and correct for it, have been demonstrated to be quite successful on such objectives. There are even simple 2D examples such as the Rosenbrock function where these methods can demonstrate considerable advantages over gradient descent. Thus it is reasonable to suspect that the deep learning problem could be resolved by the application of such techniques. Unfortu-

nately, there has yet to be a demonstration that any of these methods are effective on deep learning problems that are known to be difficult for gradient descent.

Much of the recent work on applying 2nd-order methods to learning has focused on making them practical for large datasets. This is usually attempted by adopting an "on-line" approach akin to the one used in stochastic gradient descent (SGD). The only demonstrated advantages of these methods over SGD is that they can sometimes converge in fewer training epochs and that they require less tweaking of meta-parameters, such as learning rate schedules.

The most important recent advance in learning for deep networks has been the development of layer-wise unsupervised pre-training methods (Hinton & Salakhutdinov, 2006; Bengio et al., 2007). Applying these methods before running SGD seems to overcome the difficulties associated with deep learning. Indeed, there have been many successful applications of these methods to hard deep learning problems, such as auto-encoders and classification nets. But the question remains: why does pre-training work and why is it necessary? Some researchers (e.g. Erhan et al., 2010) have investigated this question and proposed various explanations such as a higher prevalence of bad local optima in the learning objectives of deep models.

Another explanation is that these objectives exhibit pathological curvature making them nearly impossible for curvature-blind methods like gradient-descent to successfully navigate. In this paper we will argue in favor of this explanation and provide a solution in the form of a powerful semi-online 2nd-order optimization algorithm which is practical for very large models and datasets. Using this technique, we are able to overcome the under-fitting problem encountered when training deep auto-encoder neural nets far more effectively than the pre-training + fine-tuning approach proposed by Hinton & Salakhutdinov (2006). Being an optimization algorithm, our approach doesn't deal specifically with the problem of over-fitting, however we show that this is only a serious issue for one of the three deep-auto encoder problems considered by Hinton & Salakhutdinov, and can be handled by the usual methods of regularization.

These results also help us address the question of why deep-learning is hard and why pre-training sometimes

helps. Firstly, while bad local optima do exist in deep-networks (as they do with shallow ones) in practice they do not seem to pose a significant threat, at least not to strong optimizers like ours. Instead of bad local minima, the difficulty associated with learning deep auto-encoders is better explained by regions of pathological curvature in the objective function, which to 1st-order optimization methods resemble bad local minima.

2. Newton's method

In this section we review the canonical 2nd-order optimization scheme, Newton's method, and discuss its main benefits and why they may be important in the deep-learning setting. While Newton's method itself is impractical on large models due to the quadratic relationship between the size of the Hessian and the number of parameters in the model, studying it nevertheless informs us about how its more practical derivatives (i.e. quasi-Newton methods) might behave.

Newton's method, like gradient descent, is an optimization algorithm which iteratively updates the parameters $\theta \in \mathbb{R}^N$ of an objective function f by computing search directions p and updating θ as $\theta + \alpha p$ for some α. The central idea motivating Newton's method is that f can be locally approximated around each θ, up to 2nd-order, by the quadratic:

$$f(\theta + p) \approx q_\theta(p) \equiv f(\theta) + \nabla f(\theta)^\top p + \frac{1}{2} p^\top \mathrm{B} p \quad (1)$$

where $B = \mathrm{H}(\theta)$ is the Hessian matrix of f at θ. Finding a good search direction then reduces to minimizing this quadratic with respect to p. Complicating this idea is that H may be indefinite so this quadratic may not have a minimum, and moreover we don't necessarily trust it as an approximation of f for large values of p. Thus in practice the Hessian is "damped" or re-conditioned so that $\mathrm{B} = \mathrm{H} + \lambda \mathrm{I}$ for some constant $\lambda \geq 0$.

2.1. Scaling and curvature

An important property of Newton's method is "scale invariance". By this we mean that it behaves the same for any linear rescaling of the parameters. To be technically precise, if we adopt a new parameterization $\hat{\theta} = A\theta$ for some invertible matrix A, then the optimal search direction in the new parameterization is $\hat{p} = Ap$ where p is the original optimal search direction. By contrast, the search direction produced by gradient descent has the opposite response to linear re-parameterizations: $\hat{p} = A^{-\top} p$.

Scale invariance is important because, without it, poorly scaled parameters will be much harder to optimize. It also eliminates the need to tweak learning rates for individual parameters and/or anneal global learning-rates according to arbitrary schedules. Moreover, there is an implicit "scaling" which varies over the entire parameter space and is determined by the local curvature of the objective function.
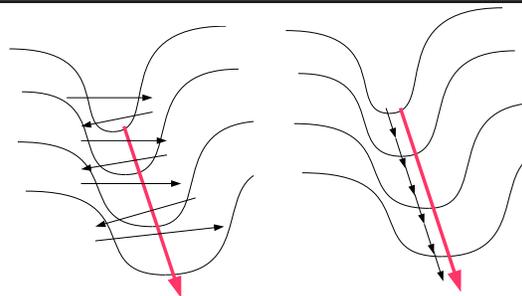

Figure 1. Optimization in a long narrow valley

By taking the curvature information into account (in the form of the Hessian), Newton's method rescales the gradient so it is a much more sensible direction to follow.

Intuitively, if the curvature is low (and positive) in a particular descent direction d, this means that the gradient of the objective changes slowly along d, and so d will remain a descent direction over a long distance. It is thus sensible to choose a search direction p which travels far along d (i.e. by making $p^\top d$ large), even if the amount of reduction in the objective associated with d (given by $-\nabla f^\top d$) is relatively small. Similarly if the curvature associated with d is high, then it is sensible to choose p so that the distance traveled along d is smaller. Newton's method makes this intuition rigorous by computing the distance to move along d as its reduction divided by its associated curvature: $-\nabla f^\top d / d^\top \mathrm{H} d$. This is precisely the point along d after which f is predicted by (1) to start increasing.

Not accounting for the curvature when computing search directions can lead to many undesirable scenarios. First, the sequence of search directions might constantly move too far in directions of high curvature, causing an unstable "bouncing" behavior that is often observed with gradient descent and is usually remedied by decreasing the learning rate. Second, directions of low curvature will be explored much more slowly than they should be, a problem exacerbated by lowering the learning rate. And if the only directions of *significant* decrease in f are ones of low curvature, the optimization may become too slow to be practical and even appear to halt altogether, creating the false impression of a local minimum. It is our theory that the under-fitting problem encountered when optimizing deep nets using 1st-order techniques is mostly due to such techniques becoming trapped in such false local minima.

Figure 1 visualizes a "pathological curvature scenario", where the objective function locally resembles a long narrow valley. At the base of the valley is a direction of low reduction and low curvature that needs to be followed in order to make progress. The smaller arrows represent the steps taken by gradient descent with large and small learning rates respectively, while the large arrow along the base of the valley represents the step computed by Newton's method. What makes this scenario "pathological" is not the presence of merely low or high curvature directions,

---

Algorithm 1 The Hessian-free optimization method

1: **for** $n = 1, 2, ...$ **do**
2:     $g_n \leftarrow \nabla f(\theta_n)$
3:     compute/adjust λ by some method
4:     define the function $B_n(d) = \mathbf{H}(\theta_n)d + \lambda d$
5:     $p_n \leftarrow$ CG-Minimize$(B_n, -g_n)$
6:     $\theta_{n+1} \leftarrow \theta_n + p_n$
7: **end for**

---

but the mixture of *both* of them together.

2.2. Examples of pathological curvature in neural nets

For a concrete example of pathological curvature in neural networks, consider the situation in which two units a and b in the same layer have *nearly* identical incoming and outgoing weights and biases. Let d be a descent direction which increases the value of one of a's outgoing weights, say parameter i, while simultaneously decreasing the corresponding weight for unit b, say parameter j, so that $d_k = \delta_{ik} - \delta_{jk}$. d can be interpreted as a direction which "differentiates" the two units. The reduction associated with d is $-\nabla f^\top d = (\nabla f)_j - (\nabla f)_i \approx 0$ and the curvature is $d^\top \mathbf{H} d = (\mathbf{H}_{ii} - \mathbf{H}_{ij}) + (\mathbf{H}_{jj} - \mathbf{H}_{ji}) \approx 0 + 0 = 0$. Gradient descent will only make progress along d which is proportional to the reduction, which is very small, whereas Newton's methods will move much farther, because the associated curvature is also very small.

Another example of pathological curvature, particular to deeper nets, is the commonly observed phenomenon where, depending on the magnitude of the initial weights, the gradients will either shrink towards zero or blow up as they are back-propagated, making learning of the weights before the last few layers nearly impossible. This difficulty in learning all but the last few layers is sometimes called the "vanishing gradients" problem and may be slightly mitigated by using heuristics to adapt the learning rates of each parameter individually. The issue here is not so much that the gradients become very small or very large absolutely, but rather that they become so *relative* to the gradients of the weights associated with units near the end of the net. Critically, the second-derivatives will shrink or blow up in an analogous way, corresponding to either very low or high curvature along directions which change the affected parameters. Newton's method thus will rescale these directions so that they are far more reasonable to follow.

3. Hessian-free optimization

The basis of the 2$^{\text{nd}}$-order optimization approach we develop in this paper is a technique known as Hessian-free optimization (HF), aka truncated-Newton, which has been studied in the optimization community for decades (e.g. Nocedal & Wright, 1999), but never seriously applied within machine learning.

In the standard Newton's method, $q_\theta(p)$ is optimized by computing the $N \times N$ matrix B and then solving the system $Bp = -\nabla f(\theta)^\top p$. This is prohibitively expensive when N is large, as it is with even modestly sized neural networks. Instead, HF optimizes $q_\theta(p)$ by exploiting two simple ideas. The first is that for an N-dimensional vector d, $\mathbf{H}d$ can be easily computed using finite differences at the cost of a single extra gradient evaluation via the identity:

$$\mathbf{H}d = \lim_{\epsilon \to 0} \frac{\nabla f(\theta + \epsilon d) - \nabla f(\theta)}{\epsilon}$$

The second is that there is a very effective algorithm for optimizing quadratic objectives (such as $q_\theta(p)$) which requires only matrix-vector products with B: the linear conjugate gradient algorithm (CG). Now since in the worst case CG will require N iterations to converge (thus requiring the evaluation of N Bd-products), it is clearly impractical to wait for CG to completely converge in general. But fortunately, the behavior CG is such that it will make significant progress in the minimization of $q_\theta(p)$ after a much more practical number of iterations. Algorithm 1 gives the basic skeleton of the HF method.

HF is appealing because unlike many other quasi-Newton methods it does not make any approximation to the Hessian. Indeed, the $\mathbf{H}d$ products can be computed accurately by the finite differences method, or other more stable algorithms. HF differs from Newton's method only because it is performing an incomplete optimization (via un-converged CG) of $q_\theta(p)$ in lieu of doing a full matrix inversion.

Another appealing aspect of the HF approach lies in the power of the CG method. Distinct from the non-linear CG method (NCG) often used in machine learning, linear CG makes strong use of the quadratic nature of the optimization problem it solves in order to iteratively generate a set of "conjugate directions" d_i (with the property that $d_i^\top A d_j = 0$ for $i \neq j$) and optimize along these independently and exactly. In particular, the movement along each direction is precisely what Newton's method would select, the reduction divided by the curvature, i.e. $-\nabla f^\top d_i / d_i^\top A d_i$, a fact which follows from the conjugacy property. On the other hand, when applying the non-linear CG method (which is done on f directly, not q_θ), the directions it generates won't remain conjugate for very long, even approximately so, and the line search is usually performed inexactly and at a relatively high expense.

Nevertheless, CG and NCG are in many ways similar and NCG even becomes equivalent to CG when it uses an exact line-search and is applied to a quadratic objective (i.e. one with constant curvature). Perhaps the most important difference is that when NCG is applied to a highly non-linear objective f, the underlying curvature evolves with each new search direction processed, while when CG is applied to the local quadratic approximation of f (i.e. q_θ), the curvature remains fixed. It seems likely that the later condition would allow CG to be much more effective than NCG

at finding directions of low reduction and curvature, as directions of high reduction and high curvature can be found by the early iterations of CG and effectively "subtracted away" from consideration via the conjugate-directions decomposition. NCG, on the other hand, must try to keep up with the constantly evolving curvature conditions of f, and therefore focus on the more immediate directions of high-reduction and curvature which arise at each successively visited position in the parameter space.

4. Making HF suitable for machine learning problems

Our experience with using off-the-shelf implementations of HF is that they simply don't work for neural network training, or are at least grossly impractical. In this section we will describe the modifications and design choices we made to the basic HF approach in order to yield an algorithm which is both effective and practical on the problems we considered. Note that none of these enhancements are specific to neural networks, and should be applicable to other optimization problems that are of interest to machine-learning researchers.

4.1. Damping

The issue of damping, like with standard Newton's method, is of vital importance to the HF approach. Unlike methods such as L-BFGS where the curvature matrix is crudely approximated, the exact curvature matrix implicitly available to the HF method allows for the identification of directions with extremely low curvature. When such a direction is found that also happens to have a reasonable large reduction, CG will elect to move very far along it, and possibly well outside of the region where (1) is a sensible approximation. The damping parameter λ can be interpreted as controlling how "conservative" the approximation is, essentially by adding the constant $\lambda\|d\|^2$ to the curvature estimate for each direction d. Using a fixed setting of λ is not viable for several reasons, but most importantly because the relative scale of B is constantly changing. It might also be the case that the "trustworthiness" of the approximation varies significantly over the parameter space.

There are advanced techniques, known as Newton-Lanczos methods, for computing the value of λ which corresponds to a given "trust-region radius" τ. However, we found that such methods were very expensive and thus not cost-effective in practice and so instead we used a simple Levenberg-Marquardt style heuristic for adjusting λ directly: **if** $\rho < \frac{1}{4}$: $\lambda \leftarrow \frac{3}{2}\lambda$ **elseif** $\rho > \frac{3}{4}$: $\lambda \leftarrow \frac{2}{3}\lambda$ **endif** where ρ is the "reduction ratio". The reduction ratio is a scalar quantity which attempts to measure the accuracy of q_θ and is given by:

$$\rho = \frac{f(\theta + p) - f(\theta)}{q_\theta(p) - q_\theta(0)}$$

4.2. Computing the matrix-vector products

While the product Hd can be computed using finite-differences, this approach is subject to numerical problems and also requires the computationally expensive evaluation of non-linear functions. Pearlmutter (1994) showed that there is an efficient procedure for computing the product Hd exactly for neural networks and several other models such as RNNs and Boltzmann machines. This algorithm is like backprop as it involves a forward and backward pass, is "local", and has a similar computational cost. Moreover, for standard neural nets it can also be performed without the need to evaluate non-linear functions.

In the development of his on-line 2$^{\text{nd}}$-order method "SMD", Schraudolph (2002) generalized Pearlmutter's method in order to compute the product Gd where G is the Gauss-Newton approximation to the Hessian. While the classical Gauss-Newton method applies only to a sum-of-squared-error objective, it can be extended to neural networks whose output units "match" their loss function (e.g. logistic units with cross-entropy error).

While at first glance this might seem pointless since we can already compute Hd with relative efficiency, there are good reasons for using G instead of H. Firstly, the Gauss-Newton matrix G is guaranteed to be positive semi-definite, even when un-damped, which avoids the problem of negative curvature, thus guaranteeing that CG will work for any positive value of λ. Mizutani & Dreyfus (2008) argue against using G and that recognizing and exploiting negative curvature is important, particularly for training neural nets. Indeed, some implementations of HF will perform a check for directions of negative curvature during the CG runs and if one is found they will abort CG and run a specialized subroutine in order to search along it. Based on our limited experience with such methods we feel that they are not particularly cost-effective. Moreover, on all of the learning problems we tested, using G instead of H consistently resulted in much better search directions, even in situations where negative curvature was not present. Another more mundane advantage of using G over H is that the associated matrix-vector product algorithm for G uses about half the memory and runs nearly twice as fast.

4.3. Handling large datasets

In general, the computational cost associated with computing the Bd products will grow linearly with the amount of training data. Thus for large training datasets it may be impractical to compute these vectors as many times as is needed by CG in order to sufficiently optimize $q_\theta(p)$. One obvious remedy is to just truncate the dataset when computing the products, but this is unsatisfying. Instead we seek something akin to "online learning", where the dataset used for each gradient evaluation is a constantly changing subset of the total, i.e. a "mini-batch".

Fortunately, there is a simple way to adapt the HF as an on-

line algorithm which we have found works well in practice, although some care must be taken. One might be tempted to cycle through a sequence of mini-batches for the evaluation of each $\mathrm{B}d$ product, but this is a bad strategy for several reasons. Firstly, it is much more efficient to compute these products if the unit activations for each training example can be cached and reused across evaluations, which would be much less viable if the set of training examples is constantly changing. Secondly, and more importantly, the CG algorithm is not robust to changes in the B matrix while it is running. The power of the CG method relies on the invariants it maintains across iterations, such as the conjugacy of its search directions. These will be quickly violated if the implicit definition of B is constantly changing as CG iterates. Thus the mini-batch should be kept constant during each CG run, cycling only at the end of each HF iteration. It might also be tempting to try using very small mini-batches, perhaps even of size 1. This strategy, too, is problematic since the B matrix, if defined using a very small mini-batch, will not contain enough useful curvature information to produce a good search direction.

The strategy we found that works best is to use relatively large mini-batches, the optimal size of which grows as the optimization progresses. And while the optimal mini-batch size may be a function of the size of the model (we don't have enough data to make a determination) it critically doesn't seem to bear any relation to the total dataset size. In our experiments, while we used mini-batches to compute the $\mathrm{B}d$ products, the gradients and log-likelihoods were computed using the entire dataset. The rationale for this is quite simple: each HF iteration involves a run of CG which may require hundreds of $\mathrm{B}d$ evaluations but only 1 gradient evaluation. Thus it is cost-effective to obtain a much higher quality estimate of the gradient. And it should be noted that unlike SGD which performs tens of thousands of iterations, the number of iterations performed by our HF approach rarely exceeds 200.

4.4. Termination conditions for CG

Implementations of HF generally employ a convergence test for the CG runs of the form $\|\mathrm{B}p + \nabla f(\theta)\|_2 < \epsilon$ where the tolerance ϵ is chosen high enough so as to ensure that CG will terminate in a number of iterations that is practical. A popular choice seems to be $\epsilon = \min(\frac{1}{2}, \|\nabla f(\theta)\|_2^{\frac{1}{2}})\|\nabla f(\theta)\|_2$, which is supported by some theoretical convergence results. In this section we will argue why this type of convergence test is bad and propose one which we have found works much better in practice.

While CG is usually thought of as an algorithm for finding a least-squares solution to the linear system $Ax = b$, it is not actually optimizing the squared error objective $\|Ax - b\|^2$. Instead, it optimizes the quadratic $\phi(x) = \frac{1}{2}x^\top Ax - b^\top x$, and is invoked within HF implementations by setting $A = \mathrm{B}$ and $b = -\nabla f(\theta)$. While $\phi(x)$ and $\|Ax - b\|^2$ have the same global minimizer, a good but sub-optimal solution for

one may actually be a terrible solution for the other. On a typical run of CG one observes that the objective function $\phi(x)$ steadily decreases with each iteration (which is guaranteed by the theory), while $\|Ax - b\|^2$ fluctuates wildly up and down and only starts shrinking to 0 towards the very end of the optimization. Moreover, the original motivation for running CG within the HF method was to minimize the quadratic $q_\theta(p)$, and not $\|\mathrm{B}p + \nabla f(\theta)\|^2$.

Thus it is in our opinion surprising that the commonly used termination condition for CG used within HF is based on $\|Ax - b\|^2$. One possible reason is that while $\|Ax - b\|^2$ is bounded below by 0, it is not clear how to find a similar bound for $\phi(x)$ that would generate a reasonable termination condition. We experimented with several obvious heuristics and found that the best one by far was to terminate the iterations once the *relative* per-iteration progress made in minimizing $\phi(x)$ fell below some tolerance. In particular, we terminate CG at iteration i if the following condition is satisfied:

$$ i > k \quad \text{and} \quad \phi(x_i) < 0 \quad \text{and} \quad \frac{\phi(x_i) - \phi(x_{i-k})}{\phi(x_i)} < k\epsilon $$

where k determines how many iterations into the past we look in order to compute an estimate of the current per-iteration reduction rate. Choosing $k > 1$ is in general necessary because, while the average per-iteration reduction in ϕ tends to decrease over time, it also displays a considerable amount of variance and thus we need to average over many iterations to obtain a reliable estimate. In all of our experiments we set $k = \max(10, 0.1i)$ and $\epsilon = 0.0005$, thus averaging over a progressively larger interval as i grows. Note that ϕ can be computed from the CG iterates at essentially no additional cost.

In practice this approach will cause CG to terminate in very few iterations when λ is large (which makes $q_\theta(p)$ easy to optimize) which is the typical scenario during the early stages of optimization. In later stages, $q_\theta(p)$ begins to exhibit pathological curvature (as long as λ is decayed appropriately), which reflects the actual properties of f, thus making it harder to optimize. In these situations our termination condition will permit CG to run much longer, resulting in a much more expensive HF iteration. But this is the price that seemingly must be paid in order to properly compensate for the true curvature in f.

Heuristics which attempt to significantly reduce the number of iterations by terminating CG early (and we tried several, such as stopping when $f(\theta + p)$ increases), provide a speed boost early on in the optimization, but consistently seem to result in worse long-term outcomes, both in terms of generalization error and overall rate of reduction in f. A possible explanation for this is that shorter CG runs are more "greedy" and do not pursue as many low-curvature directions, which seem to be of vital importance, both for reducing generalization error and for avoiding extreme-curvature scenarios such as unit saturation and

poor differentiation between units.

4.5. Sharing information across iterations

A simple enhancement to the HF algorithm which we found improves its performance by an order of magnitude is to use the search direction p_{n-1} found by CG in the previous HF iteration as the starting point for CG in the current one. There are intuitively appealing reasons why p_{n-1} might make a good initialization. Indeed, the values of B and $\nabla f(\theta)$ for a given HF iteration should be "similar" in some sense to their values at the previous iteration, and thus the optimization problem solved by CG is also similar to the previous one, making the previous solution a potentially good starting point.

In typical implementations of HF, the CG runs are initialized with the zero vector, and doing this has the nice property that the initial value of ϕ will be non-positive (0 in fact). This in turn ensures that the search direction produced by CG will always provide a reduction in q_θ, even if CG is terminated after the first iteration. In general, if CG is initialized with a non-zero vector, the initial value of ϕ can be greater than zero, and we have indeed found this to be the case when using p_{n-1}. However, judging an initialization merely by its ϕ value may be misleading, and we found that runs initialized from p_{n-1} rather than 0 consistently yielded better reductions in q_θ, even when $\phi(p_{n-1}) \gg 0$. A possible explanation for this finding is that p_{n-1} is "wrong" within the new quadratic objective mostly along the volatile high-curvature directions, which are quickly and easily discovered and "corrected" by CG, thus leaving the harder-to-find low-curvature directions, which tend to be more stable over the parameter space, and thus more likely to remain descent directions between iterations of HF.

4.6. CG iteration backtracking

While each successive iteration of CG improves the value of p with respect to the 2nd-order model $q_\theta(p)$, these improvements are not necessarily reflected in the value of $f(\theta + p)$. In particular, if $q_\theta(p)$ is untrustworthy due to the damping parameter λ being set too low or the current mini-batch being too small or unrepresentative, then running CG past a certain number of iterations can actually be harmful. In fact, the dependency of the directions generated by CG on the "quality" of B should almost certainly increase CG iterates, as the basis from which CG generates directions (called the Krylov basis) expands to include matrix-vector products with increasingly large powers of B.

By storing the current solution for p at iteration $\lceil \gamma^j \rceil$ of CG for each j (where $\gamma > 1$ is a constant; 1.3 in our experiments), we can "backtrack" along them after CG has terminated, reducing j as long as the $\lceil \gamma^{j-1} \rceil$th iterate of CG yields a lower value of $f(x + p)$ than the $\lceil \gamma^j \rceil$th. However, we have observed experimentally that, for best performance, the value of p used to initialize the next CG run (as

described in the previous sub-section) should *not* be backtracked in this manner. The likely explanation for this effect is that directions which are followed too strongly in p due to bad curvature information from an unrepresentative mini-batch, or λ being too small, will be "corrected" by the next run of CG, since it will use a different mini-batch and possibly a larger λ to compute B.

4.7. Preconditioning CG

Preconditioning is a technique used to accelerate CG. It does this by performing a linear change of variables $\hat{x} = Cx$ for some matrix C, and then optimizing the transformed quadratic objective given by $\hat{\phi}(\hat{x}) = \frac{1}{2}\hat{x}^\top C^{-\top} A C^{-1} \hat{x} - (C^{-1}b)^\top \hat{x}$. $\hat{\phi}$ may have more forgiving curvature properties than the original ϕ, depending on the value of the matrix C. To use preconditioned CG one specifies $M \equiv C^\top C$, with the understanding that it must be easy to solve $My = x$ for arbitrary x. Preconditioning is somewhat of an application specific art, and we experimented with many possible choices. One that we found to be particularly effective was the diagonal matrix:

$$M = \left[\text{diag} \left(\sum_{i=1}^{D} \nabla f_i(\theta) \odot \nabla f_i(\theta) \right) + \lambda I \right]^\alpha$$

where f_i is the value of the objective associated with training-case i, $\odot$ denotes the element-wise product and the exponent α is chosen to be less than 1 in order to suppress "extreme" values (we used 0.75 in our experiments). The inner sum has the nice interpretation of being the diagonal of the empirical Fisher information matrix, which is similar in some ways to the G matrix. Unfortunately, it is impractical to use the diagonal of G itself, since the obvious algorithm has a cost similar to K backprop operations, where K is the size of the output layer, which is large for auto-encoders (although typically not for classification nets).

5. Random initialization

Our HF approach, like almost any deterministic optimization scheme, is not completely immune to "bad" initializations, although it does tend to be far more robust to these than 1st-order methods. For example, it cannot break symmetry between two units in the same layer that are initialized with identical weights. However, as discussed in section 2.2 it has a far better chance than 1st-order methods of doing so if the weights are *nearly* identical.

In our initial investigations we tried a variety of random initialization schemes. The better ones, which were more careful about avoiding issues like saturation, seemed to allow the runs of CG to terminate after fewer iterations, mostly likely because these initializations resulted in more favorable local curvature properties. The best random initialization scheme we found was one of our own design,

"sparse initialization". In this scheme we hard limit the number of non-zero incoming connection weights to each unit (we used 15 in our experiments) and set the biases to 0 (or 0.5 for tanh units). Doing this allows the units to be both highly differentiated as well as unsaturated, avoiding the problem in dense initializations where the connection weights must all be scaled very small in order to prevent saturation, leading to poor differentiation between units.

6. Related work on 2nd-order optimization

LeCun et al. (1998) have proposed several diagonal approximations of the H and G matrices for multi-layer neural nets. While these are easy to invert, update and store, the diagonal approximation may be overly simplistic since it neglects the very important interaction between parameters. For example, in the "nearly identical units" scenario considered in section 2.2, a diagonal approximation would not be able to recognize the "differentiating direction" as being one of low curvature.

Amari et al. (2000) have proposed a 2nd-order learning algorithm for neural nets based on an empirical approximation of the Fisher information matrix (which can be defined for a neural net by casting it as a probabilistic model). Since Schraudolph's approach for computing Gd may be generalized to compute Fd, we were thus able to evaluate the possibility of using F as an alternative to G within our HF approach. The resulting algorithm wasn't able to make significant progress on the deep auto-encoder problems we considered, possibly indicating that F doesn't contain sufficient curvature information to overcome the problems associated with deep-learning. A more theoretical observation which supports the use of G over F in neural nets is that the ranks of F and G are D and DL respectively, where D is the size of the training set and L is the size of the output layer. Another observation is that G will converge to the Hessian as the error of the net approaches zero, a property not shared by F.

Building on the work of Pearlmutter (1994), Schraudolph (2002) proposed 2nd-order method called "Stochastic Meta-descent" (SMD) which uses an on-line 2nd-order approximation to f and optimizes it via updates to p which are also computed on-line. This method differs from HF in several important ways, but most critically in the way it optimizes p. The update scheme used by SMD is a form of preconditioned gradient-descent given by:

$$p_{n+1} = p_n + M^{-1}r_n \qquad r_n \equiv \nabla f(\theta_n) + Bp_n$$

where M is a diagonal pre-conditioning matrix chosen to approximate B. Using the previously discussed method for computing Bd products, SMD is able to compute these updates efficiently. However, using gradient-descent instead of CG to optimize $q_\theta(p)$, even with a good diagonal preconditioner, is an approach likely to fail because $q_\theta(p)$ will exhibit the same pathological curvature as the objective function f that it approximates. And pathological curvature was

NAME	SIZE	K	ENCODER DIMS
CURVES	20000	5000	784-400-200-100-50-25-6
MNIST	60000	7500	784-1000-500-250-30
FACES	103500	5175	625-2000-1000-500-30

the primary reason for abandoning gradient-descent as an optimization method in the first place. Moreover, it can be shown that in the batch case, the updates computed by SMD lie in the same Krylov subspace as those computed by an equal number of CG iterations, and that CG finds the optimal solution of $q_\theta(p)$ within this subspace.

These 2nd-order methods, plus others we haven't discussed, have only been validated on shallow nets and/or toy problems. And none of them have been shown to be fundamentally more effective than 1st-order optimization methods on deep learning problems.

7. Experiments

We present the results from a series of experiments designed to test the effectiveness of our HF approach on the deep auto-encoder problems considered by Hinton & Salakhutdinov (2006) (abbr. H&S). We adopt precisely the same model architectures, datasets, loss functions and training/test partitions that they did, so as to ensure that our results can be directly compared with theirs.

Each dataset consists of a collection of small grey-scale images of various objects such as hand-written digits and faces. Table 1 summarizes the datasets and associated experimental parameters, where **size** gives the size of the training set, **K** gives the size of minibatches used, and **encoder dims** gives the encoder network architecture. In each case, the decoder architecture is the mirror image of the encoder, yielding a "symmetric autoencoder". This symmetry is required in order to be able to apply H&S's pre-training approach. Note that CURVES is the synthetic curves dataset from H&S's paper and FACES is the augmented Olivetti face dataset.

We implemented our approach using the GPU-computing MATLAB package Jacket. We also re-implemented, using Jacket, the precise approach considered by H&S, using their provided code as a basis, and then re-ran their experiments using *many* more training epochs than they did, and for far longer than we ran our HF approach on the same models. With these extended runs we were able to obtain slightly better results than they reported for both the CURVES and MNIST experiments. Unfortunately, we were not able to reproduce their results for the FACES dataset, as each net we pre-trained had very high generalization error, even before fine-tuning. We ran each method until it either seemed to converge, or started to overfit (which happened for MNIST and FACES, but not CURVES). We found that since our method was much better at fitting the training data, it was thus more prone to

Table 2. Results (training and test errors)

	PT + NCG	RAND+HF	PT + HF
CURVES	0.74, 0.82	0.11, 0.20	0.10, 0.21
MNIST	2.31, 2.72	1.64, 2.78	1.63, 2.46
MNIST*	2.07, 2.61	1.75, 2.55	1.60, 2.28
FACES	-, 124	55.4, 139	-,-
FACES*	-,-	60.6, 122	-,-

overfitting, and so we ran additional experiments where we introduced an l_2 prior on the connection weights.

Table 2 summarizes our results, where **PT+NCG** is the pre-training + non-linear CG fine-tuning approach of H&S, **RAND+HF** is our Hessian-free method initialized randomly, and **PT+HF** is our approach initialized with pre-trained parameters. The numbers given in each entry of the table are the average sum of squared reconstruction errors on the training-set and the test-set. The *'s indicate that an l_2 prior was used, with strength 10^{-4} on MNIST and 10^{-2} on FACES. Error numbers for FACES which involve pre-training are missing due to our failure to reproduce the results of H&S on that dataset (instead we just give the test-error number they reported).

Figure 2 demonstrates the performance of our implementations on the CURVES dataset. Pre-training time is included where applicable. This plot is not meant to be a definitive performance analysis, but merely a demonstration that our method is indeed quite efficient.

8. Discussion of results and implications

The most important implication of our results is that learning in deep models can be achieved effectively and efficiently by a completely general optimizer without any need for pre-training. This opens the door to examining a diverse range of deep or otherwise difficult-to-optimize architectures for which there are no effective pre-training methods, such as asymmetric auto-encoders, or recurrent neural nets.

A clear theme which emerges from our results is that the HF optimized nets have much lower training error, implying that our HF approach does well because it is more effective than pre-training + fine-tuning approaches at solving the under-fitting problem. Because both the MNIST and FACES experiments used early-stopping, the training error numbers reported in Table 2 are actually much higher than can otherwise be achieved. When we initialized randomly and didn't use early-stopping we obtained a training error of 1.40 on MNIST and 12.9 on FACES.

A pre-trained initialization benefits our approach in terms of optimization speed and generalization error. However, there doesn't seem to be any significant benefit in regards to under-fitting, as our HF approach seems to solve this problem almost completely by itself. This is in stark contrast to the situation with 1st-order optimization algorithms, where the main hurdle overcome by pre-training is that of under-fitting, at least in the setting of auto-encoders.
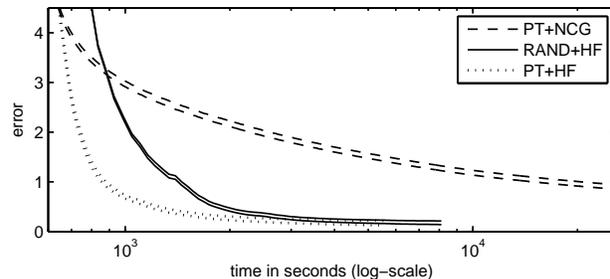

Figure 2. Error (train and test) vs. computation time on CURVES

Based on these results we can hypothesize that the way pre-training helps 1st-order optimization algorithms overcome the under-fitting problem is by placing the parameters in a region less affected by issues of pathological curvature in the objective, such as those discussed in section 2.2. This would also explain why our HF approach optimizes faster from pre-trained parameters, as more favorable local curvature conditions allow the CG runs to make more rapid progress when optimizing $q_\theta(p)$.

Finally, while these early results are very encouraging, clearly further research is warranted in order to address the many interesting questions that arise from them, such as how much more powerful are deep nets than shallow ones, and is this power fully exploited by current pre-training/fine-tuning schemes?

Acknowledgments

REFERENCES

Amari, S., Park, H., and Fukumizu, K. Adaptive method of realizing natural gradient learning for multilayer perceptrons. *Neural Computation*, 2000.

Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. Greedy layer-wise training of deep networks. In *NIPS*, 2007.

Erhan, D., Bengio, Y., Courville, A., Manzagol, P., Vincent, P., and Bengio, S. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 2010.

Hinton, G. E. and Salakhutdinov, R. R. Reducing the dimensionality of data with neural networks. *Science*, July 2006.

LeCun, Y., Bottou, L., Orr, G., and Muller, K. Efficient backprop. In Orr, G. and K., Muller (eds.), *Neural Networks: Tricks of the trade*. Springer, 1998.

Mizutani, E. and Dreyfus, S. E. Second-order stagewise backpropagation for hessian-matrix analyses and investigation of negative curvature. *Neural Networks*, 21(2-3):193 – 203, 2008.

Nocedal, J. and Wright, S. J. *Numerical Optimization*. Springer, 1999.

Pearlmutter, B. A. Fast exact multiplication by the hessian. *Neural Computation*, 1994.

Schraudolph, N. N. Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, 2002.