On optimization methods for deep learning

Quoc V. Le QUOCLE@CS.STANFORD.EDU
Jiquan Ngiam JNGIAM@CS.STANFORD.EDU
Adam Coates ACOATES@CS.STANFORD.EDU
Abhik Lahiri ALAHIRI@CS.STANFORD.EDU
Bobby Prochnow PROCHNOW@CS.STANFORD.EDU
Andrew Y. Ng ANG@CS.STANFORD.EDU
Computer Science Department, Stanford University, Stanford, CA 94305, USA

Abstract

The predominant methodology in training deep learning advocates the use of stochastic gradient descent methods (SGDs). Despite its ease of implementation, SGDs are difficult to tune and parallelize. These problems make it challenging to develop, debug and scale up deep learning algorithms with SGDs. In this paper, we show that more sophisticated off-the-shelf optimization methods such as Limited memory BFGS (L-BFGS) and Conjugate gradient (CG) with line search can significantly simplify and speed up the process of pretraining deep algorithms. In our experiments, the difference between L-BFGS/CG and SGDs are more pronounced if we consider algorithmic extensions (e.g., sparsity regularization) and hardware extensions (e.g., GPUs or computer clusters). Our experiments with distributed optimization support the use of L-BFGS with locally connected networks and convolutional neural networks. Using L-BFGS, our convolutional network model achieves 0.69% on the standard MNIST dataset. This is a state-of-the-art result on MNIST among algorithms that do not use distortions or pretraining.

1. Introduction

Stochastic Gradient Descent methods (SGDs) have been extensively employed in machine learning (Bottou, 1991; LeCun et al., 1998; Shalev-Shwartz et al., 2007; Bottou & Bousquet, 2008; Zinkevich et al., 2010). A strength of SGDs is that they are simple to implement and also fast for problems that have many training examples.

However, SGD methods have many disadvantages. One key disadvantage of SGDs is that they require much manual tuning of optimization parameters such as learning rates and convergence criteria. If one does not know the task at hand well, it is very difficult to find a good learning rate or a good convergence criterion. A standard strategy in this case is to run the learning algorithm with many optimization parameters and pick the model that gives the best performance on a validation set. Since one needs to search over the large space of possible optimization parameters, this makes SGDs difficult to train in settings where running the optimization procedure many times is computationally expensive. The second weakness of SGDs is that they are inherently sequential: it is very difficult to parallelize them using GPUs or distribute them using computer clusters.

Batch methods, such as Limited memory BFGS (L-BFGS) or Conjugate Gradient (CG), with the presence of a line search procedure, are usually much more stable to train and easier to check for convergence. These methods also enjoy parallelism by computing the gradient on GPUs (Raina et al., 2009) and/or distributing that computation across machines (Chu et al., 2007). These methods, conventionally considered to be slow, can be fast thanks to the availability of large amounts of RAMs, multicore CPUs, GPUs and computer clusters with fast network hardware.

On a single machine, the speed benefits of L-BFGS come from using the approximated second-order information (modelling the interactions between variables). On the other hand, for CG, the benefits come from using conjugacy information during optimization. Thanks to these bookkeeping steps, L-BFGS and CG

Appearing in *Proceedings of the 28th International Conference on Machine Learning*, Bellevue, WA, USA, 2011. Copyright 2011 by the author(s)/owner(s).

can be faster and more stable than SGDs.

A weakness of batch L-BFGS and CG, which require the computation of the gradient on the entire dataset to make an update, is that they do not scale gracefully with the number of examples. We found that minibatch training, which requires the computation of the gradient on a small subset of the dataset, addresses this weakness well. We found that minibatch LBFGS/CG are fast when the dataset is large.

Our experimental results reflect the different strengths and weaknesses of the different optimization methods. Among the problems we considered, L-BFGS is highly competitive or sometimes superior to SGDs/CG for low dimensional problems, especially convolutional models. For high dimensional problems, CG is more competitive and usually outperforms L-BFGS and SGDs. Additionally, using a large minibatch and line search with SGDs can improve performance.

More significant speed improvements of L-BFGS and CG over SGDs are observed in our experiments with sparse autoencoders. This is because having a larger minibatch makes the optimization problem easier for sparse autoencoders: in this case, the cost of estimating the second-order and conjugate information is small compared to the cost of computing the gradient.

Furthermore, when training autoencoders, L-BFGS and CG can be both sped up significantly (2x) by simply performing the computations on GPUs. Conversely, only small speed improvements were observed when SGDs are used with GPUs on the same problem.

We also present results showing that Map-Reduce style optimization works well for L-BFGS when the model utilizes locally connected networks (Le et al., 2010) or convolutional neural networks (LeCun et al., 1998). Our experimental results show that the speed improvements are close to linear in the number of machines when locally connected networks and convolutional networks are used (up to 8 machines considered in the experiments).

We applied our findings to train a convolutional network model (similar to Ranzato et al. (2007)) with L-BFGS on a GPU cluster and obtained 0.69% test set error. This is the state-of-the-art result on MNIST among algorithms that do not use pretraining or distortions.

Batch optimization is also behind the success of feature learning algorithms that achieve state-of-the-art performance on a variety of object recognition problems (Le et al., 2010; Coates et al., 2011) and action recognition problems (Le et al., 2011).

2. Related work

Optimization research has a long history. Examples of successful unconstrained optimization methods include Newton-Raphson's method, BFGS methods, Conjugate Gradient methods and Stochastic Gradient Descent methods. These methods are usually associated with a line search method to ensure that the algorithms consistently improve the objective function.

When it comes to large scale machine learning, the favorite optimization method is usually SGDs. Recent work on SGDs focuses on adaptive strategies for the learning rate (Shalev-Shwartz et al., 2007; Bartlett et al., 2008; Do et al., 2009) or improving SGD convergence by approximating second-order information (Vishwanathan et al., 2007; Bordes et al., 2010). In practice, plain SGDs with constant learning rates or learning rates of the form $\frac{\alpha}{\beta+t}$ are still popular thanks to their ease of implementation. These simple methods are even more common in deep learning (Hinton, 2010) because the optimization problems are nonconvex and the convergence properties of complex methods (Shalev-Shwartz et al., 2007; Bartlett et al., 2008; Do et al., 2009) no longer hold.

Recent proposals for training deep networks argue for the use of layerwise pretraining (Hinton et al., 2006; Bengio et al., 2007; Bengio, 2009). Optimization techniques for training these models include Contrastive Divergence (Hinton et al., 2006), Conjugate Gradient (Hinton & Salakhutdinov, 2006), stochastic diagonal Levenberg-Marquardt (LeCun et al., 1998) and Hessian-free optimization (Martens, 2010). Convolutional neural networks (LeCun et al., 1998) have traditionally employed SGDs with the stochastic diagonal Levenberg-Marquardt, which uses a diagonal approximation to the Hessian (LeCun et al., 1998).

In this paper, it is our goal to empirically study the pros and cons of off-the-shelf optimization algorithms in the context of unsupervised feature learning and deep learning. In that direction, we focus on comparing L-BFGS, CG and SGDs.

Parallel optimization methods have recently attracted attention as a way to scale up machine learning algorithms. Map-Reduce (Dean & Ghemawat, 2008) style optimization methods (Chu et al., 2007; Teo et al., 2007) have been successful early approaches. We also note recent studies (Mann et al., 2009; Zinkevich et al., 2010) that have parallelized SGDs without using the Map-Reduce framework.

In our experiments, we found that if we use tiled (locally connected) networks (Le et al., 2010) (which includes convolutional architectures (LeCun et al., 1998;

Lee et al., 2009a)), Map-Reduce style parallelism is still an effective mechanism for scaling up. In such cases, the cost of communicating the parameters across the network is small relative to the cost of computing the objective function value and gradient.

3. Deep learning algorithms

3.1. Restricted Boltzmann Machines

In RBMs (Smolensky, 1986; Hinton et al., 2006), the gradient used in training is an approximation formed by a taking small number of Gibbs sampling steps (Contrastive Divergence). Given the biased nature of the gradient and intractability of the objective function, it is difficult to use any optimization methods other than plain SGDs. For this reason we will not consider RBMs in our experiments.

3.2. Autoencoders and denoising autoencoders

Given an unlabelled dataset $\{x^{(i)}\}_{i=1}^{m}$, an autoencoder is a two-layer network that learns nonlinear codes to represent (or "reconstruct") the data. Specifically, we want to learn representations $h(x^{(i)}; W, b) = \sigma(Wx^{(i)} + b)$ such that $\sigma(W^T h(x^{(i)}; W, b) + c)$ is approximately $x^{(i)}$,

$$\underset{W,b,c}{\text{minimize}} \sum_{i=1}^{m} \left\| \sigma\big(W^T \sigma(Wx^{(i)} + b) + c\big) - x^{(i)} \right\|_2^2 \qquad (1)$$

Here, we use the L_2 norm to penalize the difference between the reconstruction and the input. In other studies, when x is binary, the cross entropy cost can also be used (Bengio et al., 2007). Typically, we set the activation function σ to be the sigmoid or hyperbolic tangent function.

Unlike RBMs, the gradient of the autoencoder objective can be computed exactly and this gives rise to an opportunity to use more advanced optimization methods, such as L-BFGS and CG, to train the networks.

Denoising autoencoders (Vincent et al., 2008) are also algorithms that can be trained by L-BFGS/CG.

3.3. Sparse RBMs and Autoencoders

Sparsity regularization typically leads to more interpretable features that perform well for classification. Sparse coding was first proposed by (Olshausen & Field, 1996) as a model of simple cells in the visual cortex. Lee et al. (2007); Raina et al. (2007) applied sparse coding to learn features for machine learning applications. Lee et al. (2008) combined sparsity and RBMs to learn representations that mimic certain properties of the area V2 in the visual cortex. The key idea in their approach is to penalize the deviation between the expected value of the hidden representations $\mathbb{E}\big[h_j(x; W, b)\big]$ and a preferred target activation ρ. By setting ρ to be close to zero, the hidden unit will be sparsely activated.

Sparse representations have been employed successfully in many applications such as object recognition (Ranzato et al., 2007; Lee et al., 2009a; Nair & Hinton, 2009; Yang et al., 2009), speech recognition (Lee et al., 2009b) and activity recognition (Taylor et al., 2010; Le et al., 2011).

Training sparse RBMs is usually difficult. This is due to the stochastic nature of RBMs. Specifically, in stochastic mode, the estimate of the expectation $\mathbb{E}\big[h_j(x; W, b)\big]$ is very noisy. A common practice to train sparse RBMs is to use a running estimate of $\mathbb{E}\big[h_j(x; W, b)\big]$ and penalizing only the bias (Lee et al., 2008; Larochelle & Bengio, 2008). This further complicates the optimization procedure and makes it hard to debug the learning algorithm. Moreover, it is important to tune the learning rates correctly for the different parameters W, b and c. Consequently, it can be difficult to train sparse RBMs.

In our experience, it is often faster and simpler to obtain sparse representations via autoencoders with the proposed sparsity penalties, especially when batch or large minibatch optimization methods are used.

In detail, we consider sparse autoencoders with a target activation of ρ and penalize it using the KL divergence (Hinton, 2010):

$$\sum_{j=1}^{n} D_{\mathrm{KL}}\Big(\rho \,\Big\|\, \frac{1}{m}\sum_{i=1}^{m} h_j(x^{(i)}; W, b)\Big), \qquad (2)$$

where m is the number of examples and n is the number of hidden units.

To train sparse autoencoders, we need to estimate the expected activation value for each hidden unit. However, we will not be able to compute this statistic unless we run the optimization method in batch mode. In practice, if we have a small dataset, it is better to use a batch method to train a sparse autoencoder because we do not have to tweak optimization parameters, such as minibatch size, λ as described below.

Using a minibatch of size $m' << m$, it is typical to keep a running estimate τ of the expectation $\mathbb{E}\big[h(x; W, b)\big]$. In this case, the KL penalty is

$$\sum_{j=1}^{n} D_{\mathrm{KL}}\Big(\ \rho \,\Big\|\, \lambda\frac{1}{m'}\sum_{i=1}^{m'} h_j(x^{(i)}; W, b) + (1-\lambda)\tau_j\Big) \qquad (3)$$

where λ is another tunable parameter.

3.4. Tiled and locally connected networks

RBMs and autoencoders have densely-connected network architectures which do not scale well to large images. For large images, the most common approach is to use convolutional neural networks (LeCun et al., 1998; Lee et al., 2009a). Convolutional neural networks have local receptive field architectures: each hidden unit can only connect to a small region of the image. Translational invariance is usually hard-wired by weight tying. Recent approaches try to relax this constraint (Le et al., 2010) in their tiled convolutional architectures to also learn other invariances (Goodfellow et al., 2010).

Our experimental results show that local architectures, such as tiled convolutional or convolutional architectures, can be efficiently trained with a computer cluster using the Map-Reduce framework. With local architectures, the cost of communicating the gradient over the network is often smaller than the cost of computing it (e.g., cases considered in the experiments).

4. Experiments

4.1. Datasets and computers

Our experiments were carried out on the standard MNIST dataset. We used up to 8 machines for our experiments; each machine has 4 Intel CPU cores (at 2.67 GHz) and a GeForce GTX 285 GPU. Most experiments below are done on a single machine unless indicated with "parallel."

We performed our experiments using Matlab and its GPU-plugin Jacket.[1] For parallel experiments, we used our custom toolbox that makes remote procedure calls in Matlab and Java.

In the experiments below, we report the standard metric in machine learning: the objective function evaluated on test data (i.e., test error) against time. We note that the objective function evaluated on the training shows similar trends.

4.2. Optimization methods

We are interested in off-the-shelf SGDs, L-BFGS and CG. For SGDs, we used a learning rate schedule of $\frac{\alpha}{\beta+t}$ where t is the iteration number. In our experiments, we found that it is better to use this learning rate schedule than a constant learning rate. We also use momentum, and vary the number of examples used to compute the gradient. In summary, the optimization parameters associated with SGDs are: α, β, momentum parameters (Hinton, 2010) and the number of examples in a minibatch.

[1]http://www.accelereyes.com/

We run L-BFGS and CG with a fixed minibatch for several iterations and then resample a new minibatch from the larger training set. For each new minibatch, we discard the cached optimization history in L-BFGS/CG.

In our settings, for CG and L-BFGS, there are two optimization parameters: minibatch size and number of iterations per minibatch. We use the default values[2] for other optimization parameters, such as line search parameters. For CG and LBFGS, we replaced the minibatch after 3 iterations and 20 iterations respectively. We found that these parameters generally work very well for many problems. Therefore, the only remaining tunable parameter is the minibatch size.

4.3. Autoencoder training

We compare L-BFGS, CG against SGDs for training autoencoders. Our autoencoders have 10000 hidden units and the sigmoid activation function (σ). As a result, our model has approximately 8×10^5 parameters, which is considered challenging for high order optimization methods like L-BFGS.[3]

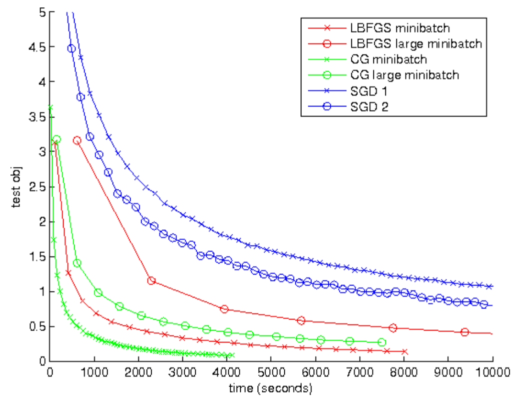

Figure 1. Autoencoder training with 10000 units on one machine.

[2]We used LBFGS in minFunc by Mark Schmidt and a CG implementation from Carl Rasmussen. We note that both of these methods are fairly optimized implementations of these algorithms; less sophisticated implementations of these algorithms may perform worse.

[3]For lower dimensional problems, L-BFGS works much better than other candidates (we omit the results due to space constraints).

For L-BFGS, we vary the minibatch size in $\{1000, 10000\}$; whereas for CG, we vary the minibatch size in $\{100, 10000\}$. For SGDs, we tried 20 combinations of optimization parameters, including varying the minibatch size in $\{1, 10, 100, 1000\}$ (when the minibatch is large, this method is also called minibatch Gradient Descent).

We compared the reconstruction errors on the test set of different optimization methods and summarize the results in Figure 1. For SGDs, we only report the results for two best parameter settings.

The results show that minibatch L-BFGS and CG with line search converge faster than carefully tuned plain SGDs. In particular, CG performs better compared to L-BFGS because computing the conjugate information can be less expensive than estimating the Hessian. CG also performs better than SGDs thanks to both the line search and conjugate information.

To understand how much estimating conjugacy helps CG, we also performed a control experiment where we tuned (increased) the minibatch size and added a line search procedure to SGDs.

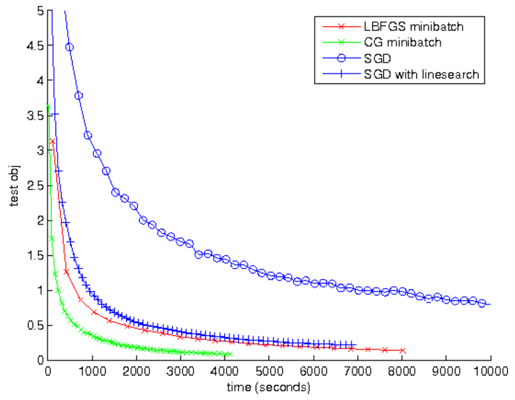

Figure 2. Control experiment with line search for SGDs.

The results are shown in Figure 2 which confirm that having a line search procedure makes SGDs simpler to tune and faster. Using information in the previous steps to form the Hessian approximation (L-BFGS) or conjugate directions (CG) further improves the results.

4.4. Sparse autoencoder training

In this experiment, we trained the autoencoders with the KL sparsity penalty. The target activation ρ is set to be 10% (a typical value for sparse autoencoders or RBMs). The weighting between the estimate for the current sample and the old estimate (λ) is set to the ratio between the minibatch size m' and 1000 ($= \min\{\frac{m'}{1000}, 1\}$). This means that our estimates of the hidden unit activations are computed by averaging over at least about 1000 examples.

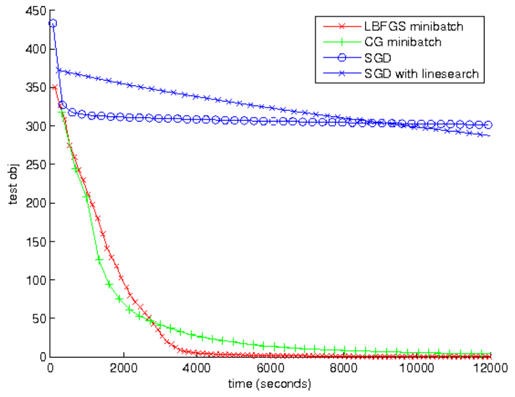

Figure 3. Sparse autoencoder training with 10000 units, $\rho = 0.1$, one machine.

We report the performance of different methods in Figure 3. The results show that L-BFGS/CG are much faster than SGDs. The difference, however, is more significant than in the case of standard autoencoders. This is because L-BFGS and CG prefer larger minibatch size and consequently it is easier to estimate the expected value of the hidden activation. In contrast, SGDs have to deal with a noisy estimate of the hidden activation and we have to set the learning rate parameters to be small to make the algorithm more stable. Interestingly, the line search does not significantly improve SGDs, unlike the previous experiment. A close inspection of the line search shows that initial step sizes are chosen to be slightly smaller (more conservative) than the tuned step size.

4.5. Training autoencoders with GPUs

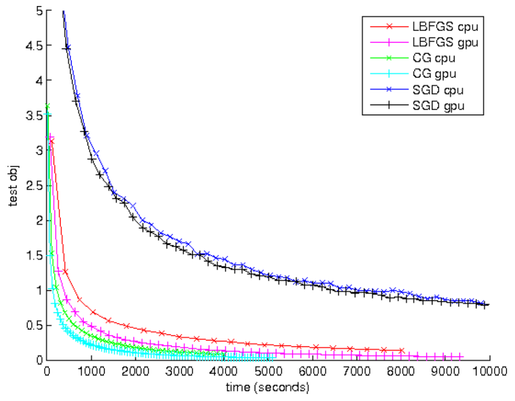

Figure 4. Autoencoder training with 10000 units, $\rho = 0.1$, one machine with GPUs. L-BFGS and CG enjoy a speed up of approximately 2x, while no significant improvement is observed for plain SGDs.

The idea of using GPUs for training deep learning algorithms was first proposed in (Raina et al., 2009). In this section, we will consider GPUs and carry out experiments with standard autoencoders to understand how different optimization algorithms perform.

Using the same experimental protocols described in 4.3, we compared optimization methods and their gains in switching from CPUs to GPUs and present the results in Figure 4. From the figure, the speed up gains are much higher for L-BFGS and CG than SGDs. This is because L-BFGS and CG prefer larger minibatch sizes which can be parallelized more efficiently on the GPUs.

4.6. Parallel training of dense networks

In this experiment, we explore optimization methods for training autoencoders in a distributed fashion using the Map-Reduce framework (Chu et al., 2007).[4] We also used the same settings for all algorithms as mentioned above in Section 4.3.

Our results with training dense autoencoders (omitted due to lack of space) show that parallelizing densely connected networks in this manner can result in slower convergence than running the method on a standalone machine. This can be attributed to the communication costs involve in passing the models and gradients across the network: the parameter vectors have a size of 64Mb, which can be a considerable amount of network traffic since it is frequently communicated.

4.7. Parallel training of local networks

If we use tiled (locally connected) networks (Le et al., 2010), Map-Reduce style gradient computation can be used as an effective way for training. In tiled networks, the number of parameters is small and thus the cost of transferring the gradient across the network can often be smaller than the cost of computing it. Specifically, in this experiment, we constrain each hidden unit to connect to a small section of the image. Furthermore, we do not share any weights across hidden units (no weight tying constraints). We learn 20 feature maps, where each map consists of 441 filters, each of size 8x8.

The results presented in Figure 5 show that SGDs are slower when a computer cluster is used. On the other hand, thanks to its preference of a larger minibatch size, L-BFGS enjoys more significant speed improvements.[5]

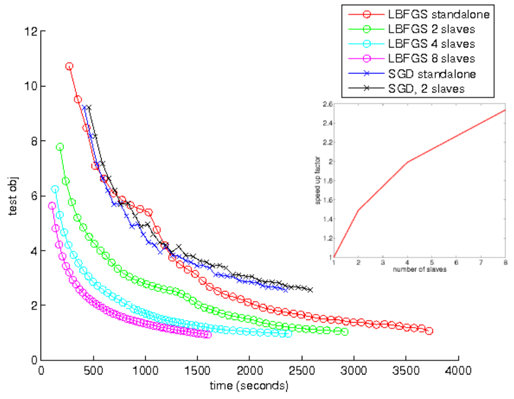

Figure 5. Parallel training of locally connected networks. With locally connected networks, the communication cost is reduced significantly. The inset figure shows the (L-BFGS) speed improvement as a function of number of slaves. The speed up factor is measured by taking the amount of time that requires each method to reach a test objective equal or better than 2.

Also, the figure shows that L-BFGS enjoys an almost linear speed up for up to 8 slave machines considered in the experiments when locally connected networks are used. On models where the number of parameters is small, L-BFGS's bookkeeping and communication cost are both small compared to gradient computations (which is distributed across the machines).

4.8. Parallel training of supervised CNNs

In this experiment, we compare different optimization methods for supervised training of two-layer convolutional neural networks (CNNs). Specifically, our model has has 16 maps of 5x5 filters in the first layer, followed by (non-overlapping) pooling units that pool over a 3x3 region. The second layer has 16 maps of 4x4 filters, without any pooling units. Additionally, we have a softmax classification layer which is connected to all the output units from the second layer. In this experiment, we distribute the gradient computations across many machines with GPUs.

The experimental results (Figure 4.8) show that L-BFGS is better than CG and SGDs on this problem because of low dimensionality (less than 10000 parameters). Map-Reduce style parallelism also significantly

[4]In detail, for parallelized methods, one central machine ("master") runs the optimization procedure while the slaves compute the objective values and gradients. At every step during optimization, the master sends the parameter across all slaves, the slaves then compute the objective function and gradient and send back to the master.

[5]In this experiment, we did not tune the minibatch size, i.e., when we have 4 slaves, the minibatch size per computer is divided by 4. We expect that tuning this minibatch size will improve the results when the number of computers goes up.

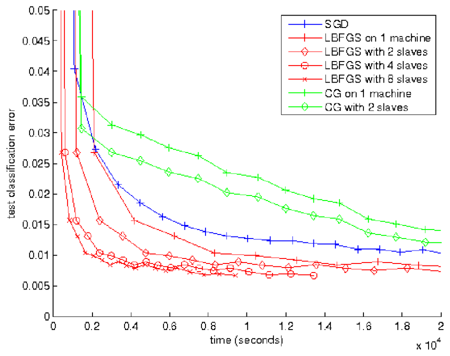

Figure 6. Parallel training of CNNs.

improves the performance of both L-BFGS and CG.

4.9. Classification on standard MNIST

Finally, we carried out experiments to determine if L-BFGS affects classification accuracy. We used a convolution network with a first layer having 32 maps of 5x5 filters and 3x3 pooling with subsampling. The second layer had 64 maps of 5x5 filters and 2x2 pooling with subsampling. This architecture is similar to that described in Ranzato et al. (2007), with the following two differences: (i) we did not use an additional hidden layer of 200 hidden units; (ii) the receptive field of our first layer pooling units is slightly larger (for computational reasons).

Table 1. Classification error on MNIST test set for some representative methods without pretraining. SGDs with diagonal Levenberg-Marquardt are used in (LeCun et al., 1998; Ranzato et al., 2007).

LeNet-5, SGDs, no distortions (LeCun et al., 1998)	0.95%
LeNet-5, SGDs, huge distortions (LeCun et al., 1998)	0.85%
LeNet-5, SGDs, distortions (LeCun et al., 1998)	0.80%
ConvNet, SGDs, no distortions (Ranzato et al., 2007)	0.89%
ConvNet, L-BFGS, no distortions (this paper)	**0.69%**

We trained our network using 4 machines (with GPUs). For every epoch, we saved the parameters to disk and used a hold-out validation set of 10000 examples[6] to select the best model. The best model is used to make predictions on the test set. The results of our method (ConvNet) using minibatch L-BFGS are reported in Table 1. The results show that the CNN, trained with L-BFGS, achieves an encouraging classification result: 0.69%. We note that this is the best result for MNIST among algorithms that do not use unsupervised pretraining or distortions. In particular, engineering distortions, typically viewed as a way to introduce domain knowledge, can improve classification results for MNIST. In fact, state-of-the-art results involve more careful distortion engineering and/or unsupervised pretraining, e.g., 0.4% (Simard et al., 2003), 0.53% (Jarrett et al., 2009), 0.39% (Ciresan et al., 2010).

[6]We used a reduced training set of 50000 examples throughout the classification experiments.

5. Discussion

In our experiments, different optimization algorithms appear to be superior on different problems. On contrary to what appears to be a widely-held belief, that SGDs are almost always preferred, we found that L-BFGS and CG can be superior to SGDs in many cases. Among the problems we considered, L-BFGS is a good candidate for optimization for low dimensional problems, where the number of parameters are relatively small (e.g., convolutional neural networks). For high dimensional problems, CG often does well.

Sparsity provides another compelling case for using L-BFGS/CG. In our experiments, L-BFGS and CG outperform SGDs on training sparse autoencoders.

We note that there are cases where L-BFGS may not be expected to perform well (e.g., if the Hessian is not well approximated with a low-rank estimate). For instance, on local networks (Le et al., 2010) where the overlaps between receptive fields are small, the Hessian has a block-diagonal structure and L-BFGS, which uses low-rank updates, may not perform well.[7] In such cases, algorithms that exploit the problem structures may perform much better.

CG and L-BFGS are also methods that can take better advantage of the GPUs thanks to their preference of larger minibatch sizes. Furthermore, if one uses tiled (locally connected) networks or other networks with a relatively small number of parameters, it is possible to compute the gradients in a Map-Reduce framework and speed up training with L-BFGS.

Acknowledgments: We thank Andrew Maas, Andrew Saxe, Quinn Slack, Alex Smola and Will Zou for comments and discussions. This work is supported by the DARPA Deep Learning program under contract number FA8650-10-C-7020.

References

Bartlett, P., Hazan, E., and Rakhlin, A. Adaptive online gradient descent. In *NIPS*, 2008.

Bengio, Y. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2009.

Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H.

[7]Personal communications with Will Zou.

Greedy layerwise training of deep networks. In *NIPS*, 2007.

Bordes, A., Bottou, L., and Gallinari, P. SGD-QN: Careful quasi-newton stochastic gradient descent. *JMLR*, 2010.

Bottou, L. Stochastic gradient learning in neural networks. In *Proceedings of Neuro-Nîmes 91*, 1991.

Bottou, L. and Bousquet, O. The tradeoffs of large scale learning. In *NIPS*. 2008.

Chu, C.T., Kim, S. K., Lin, Y. A., Yu, Y. Y., Bradski, G., Ng, A. Y., and Olukotun, K. Map-Reduce for machine learning on multicore. In *NIPS 19*, 2007.

Ciresan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. Deep big simple neural nets excel on handwritten digit recognition. *CoRR*, 2010.

Coates, A., Lee, H., and Ng, A. Y. An analysis of single-layer networks in unsupervised feature learning. In *AISTATS 14*, 2011.

Dean, J. and Ghemawat, S. Map-Reduce: simplified data processing on large clusters. *Comm. ACM*, 2008.

Do, C.B., Le, Q.V., and Foo, C.S. Proximal regularization for online and batch learning. In *ICML*, 2009.

Goodfellow, I., Le, Q.V., Saxe, A., Lee, H., and Ng, A.Y. Measuring invariances in deep networks. In *NIPS*, 2010.

Hinton, G. A practical guide to training restricted boltzmann machines. Technical report, U. of Toronto, 2010.

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

Hinton, G. E., Osindero, S., and Teh, Y.W. A fast learning algorithm for deep belief nets. *Neu. Comp.*, 2006.

Jarrett, K., Kavukcuoglu, K., Ranzato, M.A., and LeCun, Y. What is the best multi-stage architecture for object recognition? In *ICCV*, 2009.

Larochelle, H. and Bengio, Y. Classification using discriminative restricted boltzmann machines. In *ICML*, 2008.

Le, Q. V., Ngiam, J., Chen, Z., Chia, D., Koh, P. W., and Ng, A. Y. Tiled convolutional neural networks. In *NIPS*, 2010.

Le, Q. V., Zou, W., Yeung, S. Y., and Ng, A. Y. Learning hierarchical spatio-temporal features for action recognition with independent subspace analysis. In *CVPR*, 2011.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient based learning applied to document recognition. *Proceeding of the IEEE*, 1998.

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

Lee, H., Battle, A., Raina, R., and Ng, Andrew Y. Efficient sparse coding algorithms. In *NIPS*, 2007.

Lee, H., Ekanadham, C., and Ng, A. Y. Sparse deep belief net model for visual area V2. In *NIPS*, 2008.

Lee, H., Grosse, R., Ranganath, R., and Ng, A.Y. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *ICML*, 2009a.

Lee, H., Largman, Y., Pham, P., and Ng, A. Y. Unsupervised feature learning for audioclassification using convolutional deep belief networks. In *NIPS*, 2009b.

Mann, G., McDonald, R., Mohri, M., Silberman, N., and Walker, D. Efficient large-scale distributed training of conditional maximum entropy models. In *NIPS*, 2009.

Martens, J. Deep learning via hessian-free optimization. In *ICML*, 2010.

Nair, V. and Hinton, G. E. 3D object recognition with deep belief nets. In *NIPS*, 2009.

Olshausen, B. and Field, D. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, 1996.

Raina, R., Battle, A., Lee, H., Packer, B., and Ng, A.Y. Self-taught learning: Transfer learning from unlabelled data. In *ICML*, 2007.

Raina, R., Madhavan, A., and Ng, A. Y. Large-scale deep unsupervised learning using graphics processors. In *ICML*, 2009.

Ranzato, M., Huang, F. J, Boureau, Y., and LeCun, Y. Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *CVPR*, 2007.

Shalev-Shwartz, S., Singer, Y., and Srebro, N. Pegasos: Primal estimated sub-gradient solver for svm. In *ICML*, 2007.

Simard, P., Steinkraus, D., and Platt, J. Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR*, 2003.

Smolensky, P. Information processing in dynamical systems: foundations of harmony theory. In *Parallel distributed processing*, 1986.

Taylor, G.W., Fergus, R., Lecun, Y., and Bregler, C. Convolutional learning of spatio-temporal features. In *ECCV*, 2010.

Teo, C. H., Le, Q. V., Smola, A. J., and Vishwanathan, S. V. N. A scalable modular convex solver for regularized risk minimization. In *KDD*, 2007.

Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P. A. Extracting and composing robust features with denoising autoencoders. In *ICML*, 2008.

Vishwanathan, S. V. N., Schraudolph, N. N., Schmidt, M. W., and Murphy, K. P. Accelerated training of conditional random fields with stochastic gradient methods. In *ICML*, 2007.

Yang, J., Yu, K., Gong, Y., and Huang, T. Linear spatial pyramid matching using sparse coding for image classification. In *CVPR*, 2009.

Zinkevich, M., Weimer, M., Smola, A., and Li, L. Parallelized stochastic gradient descent. In *NIPS*, 2010.