Optimization Properties of Deep Networks

CS229T/STATS231 Winter 2014 – Final Report

Awni Hannun
awni@stanford.edu

Peng Qi
pengqi@stanford.edu

1 Introduction

Multilayer neural networks and other deep models have gained considerable traction over the past several years demonstrating state-of-the-art performance in many applications. Correspondingly, much work has been done to improve methods for optimizing these functions. This has proven difficult due to the highly nonconvex landscape of the typical multilayer objective and the supposed ill-conditioned objectives. Given growth in dataset sizes and that these highly expressive models tend to be used in the large data setting, stochastic optimization methods are common. However, stochastic methods tend to come with a host of hyperparameters and much tuning is involved to get these to work well which has lead to general disagreement as to the supieriority of a particular algorithm.

Until recently stochastic gradient descent (SGD) with momentum has been the standard stochastic optimizer used with deep neural networks (DNNs) [2]; however, other stochastic methods are beginning to be adopted including AdaGrad [1] and variations as well as the formulation of Nesterov's accelerated gradient (NAG) as presented in [8]. To the best of our knowledge, we do not know of any work which gives a thorough comparison of these algorithms and demonstrates when one or the other should be applied. There has been some work in automating the hyperparamter tuning process for a given algorithm [7], yet this does not answer the question of which algorithm should be used in the first place and furthermore still requires the training of many models which can be prohibitive.

Furthermore, recent work has begun to attempt an analytical understanding of these types of heirarchichal models; however, often the results are derived using simplifications or modifications of a typical DNN which make them not applicable in practice [6].

We attempt to prescribe a stochastic optimization procedure for the typical multilayer neural network objective. We develop a deeper understanding of the most common objectives and architectures used in this setting. We apply this analysis to motivate the selection of optimization procedures we study and attempt to give a thorough comparison of a few competing algorithms using existing theory and empirical results.

2 Optimization

We consider two families of stochastic optimization algorithms. Traditionally used in optimizing deep neural networks, the first class is that of momentum based methods which have recently been generalized and included the broader group of accelerated gradient algorithms [8]. The second class includes variations on the adaptive gradient algorithms as presented in [1].

2.1 Accelerated Gradient

The accelerated gradient methods are a generalization of the basic stochastic gradient descent (SGD) optimization procedure. Let $\ell(\theta)$ be the loss function of the network with parameters θ. We use accelerated SGD methods that update the parameters at time t according to

$$\begin{aligned} v_{t+1} &= \mu v_t - \eta \nabla \ell(\theta_t + \gamma v_t) \\ \theta_{t+1} &= \theta_t + v_{t+1} \end{aligned} \tag{1}$$

where η is the learning rate, and the parameter μ, known as the momentum parameter, dictates how much gradient history we take into account at every update. If we set $\mu = 0$ we recover plain SGD, and setting $\mu = 1$ uses the full gradient history at every update. The γ parameter is either active and set as $\gamma = \mu$ or inactive and set as $\gamma = 0$. When γ is inactive, we have typical SGD with "momentum", also known as classical momentum (CM). On the other hand, when γ is active the procedure is known as Nesterov's accelerated graient (NAG).

A common belief is that network objectives likely suffer from long narrow ravines leading towards local optima surrounded with walls of high curvature. This hypothesis motivates the use of momentum to encourage persistent directions of travel along the basin and suppress unwanted oscillation.

2.2 Adaptive Gradient

We study modifications to the adaptive gradient algorithm (AdaGrad) with the diagonal preconditioning matrix G. The update can be written as

$$\begin{aligned} G_{t+1} &= G_t + \mathrm{diag}(\nabla \ell(\theta_t))^2 \\ \theta_{t+1} &= \theta_t - G_{t+1}^{-1/2} \nabla \ell(\theta_t) \end{aligned} \tag{2}$$

In some sense AdaGrad achieves a similar affect as NAG by penalizing oscillating directions in which we take large steps and encouraging directions with small but consistent gradients. However, in other aspects the optimization routines behave quite differently. A simple property of AdaGrad which as we show later can drasitcally affect optimization is the nondecreasing monotinicity of the matrix G. This leads AdaGrad to penalize large yet consistent directions of travel in the optimization process. This proves locally favorable in most of the models studied in the next section, but globally the behaviour results in better performance of NAG routine over the vanilla AdaGrad algorithm.

This motivates two simple variations of the AdaGrad algorithm. The first is to replace the square root on the matrix G_t with a function which grows less quickly (e.g. cube root). The second is to decay G_t by a factor of $\gamma \in (0, 1]$ before including it in the update for G_{t+1}. This is close to the AdaDelta algorithm presented in [10], but there they take a convex combination of the two terms in the update of G_{t+1} whereas here we only decay the gradient history term.

3 Experiments

3.1 Setup

In most experiments performed, we use a standard multilayer neural network with dense connections between each layer of hidden activations. We study two classes of networks. The first class is the standard densely connected classification network trained with the cross entropy loss. For a single training input and label $(x^{(i)}, y^{(i)})$, the loss is given by

$$\ell(x^{(i)}, y^{(i)}) = -\sum_{k=1}^{K} \mathbb{I}\{y^{(i)} = k\} \log p_{i,k} \tag{3}$$

where K is the number of classes and $p_{i,k}$ is the probability the model assigns to example i taking on label k. We also study the autoencoder network which attempts to reconstruct the input after nonlinearly projecting it to a much lower dimension. We use the squared loss to train this network, which on input $x^{(i)}$ is given by

$$\ell(x^{(i)}) = \frac{1}{2} \| x^{(i)} - h(x^{(i)}) \|_2^2 \tag{4}$$

and h is the function which maps the network input to output.

The number of hidden layers and size of the networks vary; for the MNIST and CIFAR classification experiments described below we use between 2 - 4 hidden layers with 200 units each. The autoencoder is only trained on MNIST, and we use the same 1.6 million parameter, 7 hidden layer network as in [3] which is known to be a difficult to optimize benchmark. The leaky rectified linear nonlinearity ($f(x) = \max\{0.01x, x\}$) is applied at each hidden unit. Most computation is accelerated using GPUs and the Gnumpy library for Python [9].

We experiment mostly with the MNIST handwritten digits dataset which consists of 60,000 training images and 10,000 test images. The images are all 28×28 pixels; however, for computational efficiency, prior to training classification networks we project the data onto the leading principal components (between 50 and 100) and use this as the input. We also ran experiments on the CIFAR-10 object recognition dataset [4]. This benchmark consists of 50,000 training images and 10,000 test images all 32×32 and RGB. We first grayscale the images then use the same preprocessing as that of the MNIST data.

3.2 Results

As a preliminary, we grid searched over all learning rate and momentum hyperparameters to select the best for each optimization strategy including vanilla SGD and CM. The standard SGD and CM (e.g. $\gamma = 0$ in 1) optimization algorithms did not converge nearly as quickly as variations of AdaGrad and the NAG optimizers, thus we restrict our attention to those two in the following results and analysis.

Figure 1: Learning curves for the classification network on MNIST. The network has 4 hidden layers each with 200 hidden units. The y-axis displays a windowed average of the crossentropy cost after each parameter update. Ada3 is the cube root version of AdaGrad as discussed in section 2.2.

Figure 2: Learning curves for the classification network on CIFAR-10. Details are the same as Figure 1.

In both classification networks trained on MNIST and CIFAR-10 (cf. Figures 1 and 2 respectively), NAG and standard AdaGrad perform comparably well. However, by choosing γ wisely for the decayed version of AdaGrad, we can achieve much faster convergence particularly after the initial stages of fast learning. The oscillations in Figure 2 are somewhat inexplicable. For the NAG updater these oscillations could be due to the same observations as [5], suggesting that the momentum is on the high side. However, we observe fastest convergence with the momentum set approximately in the interval $\mu \in [0.9, 0.99]$. This also suggests that an adaptive restart of the velocity v_t, (i.e. resetting the velocity to zero on some indicator such as an increase in the objective function) could achieve faster convergence [5].

Figure 3: Learning curves for the autoencoder used in [3]. The network has 7 hidden layers of sizes 1000, 500, 250, 30, 250, 500, and 1000 with 1.6 million trainable parameters. The y-axis is a windowed average of mean-squared reconstruction error, the objective used to train the network.

In order to robustly test the decayed version of the AdaGrad algorithm, we trained an autoencoder with 7 hidden layers. Again, with the correct setting of γ in the decayed version of AdaGrad, we notice much faster convergence, although the difference between NAG is not as pronounced in the later stages of learning. The number of iterations needed to train this network causes the vanishing learning rate problem for AdaGrad to be more severe even for the cubic version.

Figures xxx in the appendix demonstrate another nice property of the decayed version of AdaGrad, namely robustness to learning rate. Notice that in Figure xx if we do not tune the learning rate well, AdaGrad will simply not converge and NAG (cf. Figure xx), while slightly more robust, will take a long time. However, as demonstrated in Figure xx, the decayed AdaGrad enjoys a robustness to a wide range of learning rates, converging quickly to nearly the same point at the same rate after the initial stages of learning.

4 Analysis

5 Conclusion

References

[1] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 2011.

[2] G. Hinton. A practical guide to training restricted boltzmann machines. *Technical Report*, 2010.

[3] G. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[4] A. Krizhevsky. Learning multiple layers of features from tiny images. *Technical Report*, 2009.

[5] B. O'Donoghue and E. Candes. Adaptive restart for accelerated gradient schemes. *Technical Report*, 2012.

[6] A. Saxe, J. McClelland, and S. Ganguli. Dynamics of learning in deep linear neural networks. *NIPS Workshop on Deep Learning*, 2013.

[7] J. Snoek, H. Larochelle, and R. Adams. Practical bayesian optimization of machine learning algorithms. *Neural Information Processing Systems*, 2012.

[8] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of momentum and initialization in deep learning. *International Conference on Machine Learning*, 2013.

[9] T. Tielemen. Gnumpy: an easy way to use GPU boards in Python. *Technical Report*, 2010.

[10] M. Zeiler. AdaDelta: An adaptive learning rate method. *Technical Report*, 2012.

A Further Experiments

Figure 4: Learning curves for the AdaGrad optimizer with different global step size η. The autoencoder is the same as in Figure 3.

Figure 5: Learning curves for the Nesterov optimizer with different global step sizes η and a fixed momentum of $\mu = 0.99$. The autoencoder is the same as in Figure 3.

Figure 6: Learning curves for the decayed version of AdaGrad, $\gamma = 0.99$, with different global step sizes η. The autoencoder is the same as in Figure 3.