Faster gradient descent via an adaptive learning rate

Mathieu Ravaut*
CentraleSupelec
University of Toronto
27 King's College Circle
Toronto, ON M5S
mravox@cs.toronto.edu

Satya Krishna Gorti*
University of Toronto
27 King's College Circle
Toronto, ON M5S
satyag@cs.toronto.edu

Abstract

Any gradient descent requires to choose a learning rate. With deeper and deeper models, tuning that learning rate can easily become tedious and does not necesarily lead to an ideal convergence. We propose a variation of the gradient descent algorithm in the which the learning rate η is not fixed. Instead, we learn η itself, either by another gradient descent (first-order method), or by Newton's method (second-order). That way, gradient descent for any machine learning algorithm can be optimized.

1 Introduction

In the past decades, gradient descent has been widely adopted to optimize the loss function in machine learning algorithms. Lately, the machine learning community has also used the stochastic gradient descent alternative, which processes input data in batches. Gradient descent can be used in any dimension, and presents the advantage of being easy to understand and inexpensive to compute. Under certain assumptions on the loss function (such as convexity), gradient descent is guaranteed to converge to the minimum of the function, *with a carefully chosen learning rate* η. Moreover, stochastic gradient descent has proven to be very efficient even in situations where the loss function is not convex, as is mostly the case with modern deep neural networks. Other methods such as Newton's method guarantee a much faster convergence, but are typically very expensive. Newton's method for instance requires to compute the inverse of the Hessian matrix of the loss function with regards to all parameters, which is impossible with today's hardware and today's deep networks using millions of parameters [8].

The quality of a gradient descent heavily depends on the choice of the learning rate η. A too high learning rate will see the loss function jumping around the direction of steepest descent, and eventually diverge. While a very low learning rate guarantees non-divergence, convergence will be very slow, and the loss function might get stuck in a local minimum. Choosing an ideal learning rate requires an intuition of the problem. Typically, researchers would start by performing a random search or a line-search over a set of different orders of magnitude of learning rates, but this is long and costly. Besides, these methods assume a fixed learning rate over time, as doing one search per iteration would require exponential computation cost. Usually, we see researchers setting the learning rate to an initial value then decreasing it one or a few items after training has progressed.

In this paper, we propose to automatically find the learning rate at each epoch. We still need to input an initial value, but at each iteration, our model will find a learning rate that optimizes best the loss function at this point of learning. We explore a first-order method and a second-order one to do so, with a strong emphasis on the latter. Our method could be applied to any machine learning algorithm using gradient descent. We show faster initial convergence on a variety of tasks and models, including:

* indicates equal contribution

- Linear regression on the Boston Housing Prices dataset
- Logistic regression on MNIST
- Image classification with neural networks on Cifar-10 and Cifar-100

2 Related work

Techniques for improving optimization of machine learning problems have a long history. The most commonly used technique is Stochastic Gradient Descent (SGD), combined with tricks like momentum [11]. While using SGD with complex neural network architectures, the non-convex nature of the problem makes it even harder to optimize.

Recently used alternatives to SGD include Contrastive Divergence [5], Conjugate Gradients [6], Stochastic Diagonal Levenberg-Marquardt [9], and Hessian-free optimization [10]. All these techniques optimize the parameters of the model with the objective of minimizing a loss function. To the best of our knowledge, none of these techniques have been applied to optimize the learning rate at every iteration as we will discuss.

In practice, techniques of the form $\frac{\eta}{\beta+t}$ that decay the learning rate over time are popular. The Adagrad method [3] for instance divides the learning rate at each step by the norm of all previous gradients. Adadelta [14] is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Adam [7] is built on top of these methods. It stores an exponentially decaying average of past squared gradients like Adadelta, and also keeps an exponentially decaying average of past gradients, similar to momentum.

LeCun et al. [9] mention that the optimal learning rate is $\eta_{opt} = \frac{1}{\lambda_{max}}$ where λ_{max} is the maximum eigenvalue of the Hessian matrix of the loss function with regards to all parameters. With today's neural network architectures, computing the Hessian is a computationally very expensive operation. Lecun et al. avoid computing the Hessian via approximating its eigenvalues while Martens et al. [10] estimate the Hessian via finite differences.

Tom Schaul et al. [12] model the expected loss after a SGD update to deduce the best learning rate update *for each dimension*. To do so, they assume a diagonal Hessian matrix. In practice, this is evidently not the case, and they rely on observed samples of the gradient to update the learning rate.

Similar to Martens et al. [10] we use finite differences to approximate first order and second order derivatives of the loss function for updating the learning rate at every iteration.

3 Adaptive learning rate

SGD with an adaptive learning rate has the following form:

$$w(t+1) = w(t) - \eta(t)g(t) \tag{1}$$
$$\eta(t+1) = h(\eta(t)) \tag{2}$$

where $g(t) = \nabla L(w(t))$, L is the loss function, $w(t)$ represents the state of the model's weights at time t, and h is a *continuous* function. In the following we present several ways to update $\eta(t)$.

3.1 First-order method

The first-order method consists in doing gradient descent on the learning rate.
Let us introduce a function that will be useful in the following:

$$f : R^n \rightarrow R \tag{3}$$
$$\eta \rightarrow L(w(t) - \eta g(t)) \tag{4}$$

n corresponds to the number of trainable parameters in the model. f represents what the loss would be if we were to perform a gradient descent update with the given learning rate η.

The first-order method is written:

$$w(t+1) = w(t) - \eta(t)g(t) \tag{5}$$
$$\eta(t+1) = \eta(t) - \alpha f'(\eta(t)) \tag{6}$$

This method introduces a new "meta" learning rate α. It has the advantage of being light in cost as we only need gradients (first-order derivatives) of the loss function L. Indeed:

$$\forall \eta, f'(\eta) = -g(t)^T.\nabla L(w(t) - \eta g(t)) \text{ where . is the dot product in dimension } n \tag{7}$$

In particular, at the value of η used to get $w(t+1)$, we have the handful analytical formula:

$$f'(\eta(t)) = -g(t)^T.g(t+1) \tag{8}$$

However, in practice, we prefer to compute the first-order method with finite differences.

3.2 Second-order method

The previous method transfers the problem of choosing an ideal learning rate for weights to choosing an ideal learning rate for the learning rate itself. To avoid that problem, the second-order method uses a Newton-Raphson algorithm:

$$w(t+1) = w(t) - \eta(t)g(t) \tag{9}$$
$$\eta(t+1) = \eta(t) - \frac{f'(\eta(t))}{f''(\eta(t))} \tag{10}$$

Now all the learning only depends on the loss, there is no more constant value to choose.
However, the second derivative of f requires building the loss Hessian matrix H_L:

$$\forall \eta, f''(\eta) = g(t)^T.H_L(w(t) - \eta g(t)) \tag{11}$$

We propose to approximate this Hessian using finite differences. By using this approach on f' then f", we get an update formula of the learning rate depending only on several values of the loss function:

$$f'(\eta + \epsilon) \approx \frac{f(\eta + 2\epsilon) - f(\eta)}{2\epsilon} \tag{12}$$

$$\text{and } f'(\eta - \epsilon) \approx \frac{f(\eta) - f(\eta - 2\epsilon)}{2\epsilon} \tag{13}$$

$$\text{so } f''(\eta) \approx \frac{f(\eta + 2\epsilon) + f(\eta - 2\epsilon) - 2f(\eta)}{4\epsilon^2} \tag{14}$$

Given the finite differences on the first derivative:

$$f'(\eta) \approx \frac{f(\eta + \epsilon) - f(\eta - \epsilon)}{2\epsilon} \tag{15}$$

We get the final simple formula for η:

$$\eta(t+1) = \eta(t) - 2\epsilon \frac{(f(\eta + \epsilon) - f(\eta - \epsilon))}{f(\eta + 2\epsilon) + f(\eta - 2\epsilon) - 2f(\eta)} \tag{16}$$

Assuming a constant sign for the numerator, this formula means the following: when slightly increasing the learning rate corresponds to a lower loss than slightly reducing it, then the numerator is negative. In consequence, the learning rate is raised at this update, as pushing in the ascending direction for the learning rate seems to help reducing the loss. However, reality is more complex as the denominator's sign changes as well.

4 Experiments

4.1 Practical considerations

With the second-order method, the denominator in the learning rate update might underflow. To avoid such a situation, we add to the denominator a small smoothing value when it reaches zero machine, but only when it does so:

$$\text{if } (f(\eta + 2\epsilon) + f(\eta - 2\epsilon) - 2f(\eta)) \approx 0 \tag{17}$$

$$\text{then } \eta(t+1) = \eta(t) - 2\epsilon \frac{(f(\eta + \epsilon) - f(\eta - \epsilon))}{f(\eta + 2\epsilon) + f(\eta - 2\epsilon) - 2f(\eta) + \delta} \tag{18}$$

A typical value of δ is 10^{-6}.

Besides, when updating the loss and the learning rate, one should be very careful about the order of the operations. At the *k-th* iteration, we have a loss value $L^{(k)}$ and a learning rate value $\eta^{(k)}$. Then the *(k+1)-th* step of our algorithm is written as follows:

- Get the five loss values $f(\eta^{(k)} + \epsilon)$, $f(\eta^{(k)} - \epsilon)$, $f(\eta^{(k)} + 2\epsilon)$, $f(\eta^{(k)} - 2\epsilon)$ and $f(\eta^{(k)})$
- $L^{(k+1)} \leftarrow f(\eta^{(k)})$
- $\eta^{(k+1)} \leftarrow \eta^{(k)} - 2\epsilon \frac{(f(\eta^{(k)}+\epsilon) - f(\eta^{(k)}-\epsilon))}{f(\eta^{(k)}+2\epsilon) + f(\eta^{(k)}-2\epsilon) - 2f(\eta^{(k)})}$

Note that each update of the learning rate requires five forward passes in the network to get each of the loss values. Theses forward passes are then "canceled" so that the network can come back to its standard state of the beginning of step k. This cancellation is our algorithm's trickiest part to implement.

Finally, both first-order and second-order updates of the learning rate don't guarantee that it does not reach negative values. This would be inconvenient as a gradient ascent does not reduce the loss. Therefore, we force the learning rate to stay positive after each update:

$$\eta(t+1) = max(\eta(t+1), \delta) \text{ where } \delta \text{ is the smoothing value} \tag{19}$$

In practice, the learning rate almost never reaches δ with the second-order method, but this can happen sometimes with the first-order method.

4.2 Results

We compare our second-order method to a standard SGD method, which uses a fixed learning rate schedule. In the following, this standard method will be called the *basic method*. That means, the basic method uses plain stochastic gradient descent with a fixed learning rate as optimizer. The first order method was quite unstable and less efficient. When relevant, we also include its results. We compare loss and accuracy on both training and test sets. Our goal in these experiments is not to break state of the art on the given problem, but to show an improvement in our method compared to the basic approach, especially in the first few dozens passes through the training dataset. Thus, we do not use any particular trick (not even momentum on the weights update) to improve training accuracy (like data augmentation or parameter tuning), except when we add dropout to prevent overfitting in some cases.

4.2.1 Linear regression

We first tried our adaptive learning rate schedule on linear regression applied to the Boston Housing dataset [1]. This dataset of 506 points with 13 features is suitable for a simple linear regression model. We split this dataset into 400 points for training, and the remaining 106 for test.

The model consists in fitting a vector of parameters W to the dataset. We are minimizing the square loss $L = ||X.W - Y||^2$.

Given the size of the dataset, we perform gradient descent with a full batch. After line search, a learning rate of 10^{-6} appears to offer an optimal loss reduction.

Again, the small training size leads us to use the analytical formula for the first-order learning rate update. The derivative $f'(\eta(t))$ is just the dot product between the current loss gradient and its previous value. We have two parameters to set: the initial learning rate η_{ini} and the meta-learning rate α. Common sense says that an α larger than η_{ini} might get the learning rate negative, so we look for an α smaller than η_{ini}. After a double line-search, we use the two values $\eta_{ini} = 10^{-7}$ and $\alpha = 10^{-9}$.

For the second-order method, we use the finite-differences approximation previously described. Here as well, the training set is considered as one large batch. We set $\epsilon = 10^{-5}$, so we now just need to choose an initial value η_{ini}. A value of $\eta_{ini} = 10^{-4}$ gave a remarkably fast convergence.

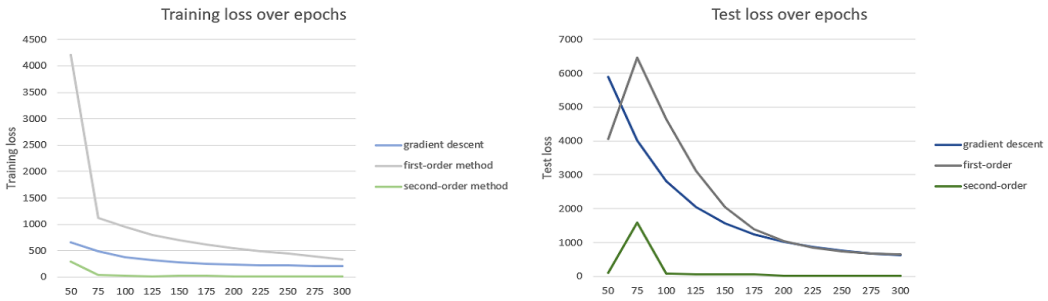

Figure 1: Basic, first-order, and second-order methods training and test losses over time

From these first results, it seems that the second-order method is much more efficient, converging after only 100 iterations versus 500 hundred (not plotted above) for the basic method. The first-order method is slower and gives a higher final training loss. Instead of computing other metrics for this task, we prefer to move on to another type of machine learning problem: logistic regression.

4.2.2 Logistic regression

We also tried our method with multi-class logistic regression on the fashion MNIST dataset [13].

4.2.3 Image classification with neural networks

CIFAR-10

A common benchmark in machine learning is to classify images on the CIFAR-10 dataset [2]. Deep neural networks are most suited for this task [8]. In consequence, we used a LeNet with 5 layers (2 convolutional followed by 3 fully-connected layers) and ResNet-18 [4]. We present results with the second-order method with both networks, then the first-order one with ResNet.

Second-order method

We first started by optimizing the LeNet model. We trained it for different learning rate values, and 0.01 appeared to be the best choice. As a plain SGD optimization without momentum tended to overfit, we introduced two dropout layers after the first two fully-connected layers. Thus, we compare the basic method with the second-order adaptive one starting at 0.01, using a batch-size of 32:

From here we can notice four things:

- The adaptive method is noisier than simple SGD.
- Training is faster with the adaptive method. Looking at the first epochs shows that the training and test losses get optimized much quicklier than in the basic case.
- Given the values that the training loss and accuracy reach, the adaptive method is probably overfitting (and the basic method seems to be on its way to overfit).

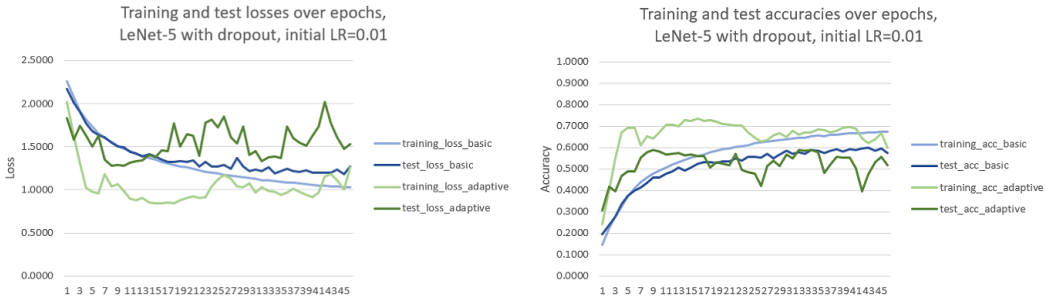

Figure 2: Second-order and basic methods with LeNet, initial learning rate: 0.01

- Even with this overfitting, the adaptive method plateaus around the same test accuracy (59%) than the basic method. This maximum test accuracy is reached in less than 10 epochs, versus around 35 epochs for the basic method.

We now look at the performance of ResNet-18. We train it with the reference learning rate of 0.1 given by [4], and a batch size of 256:

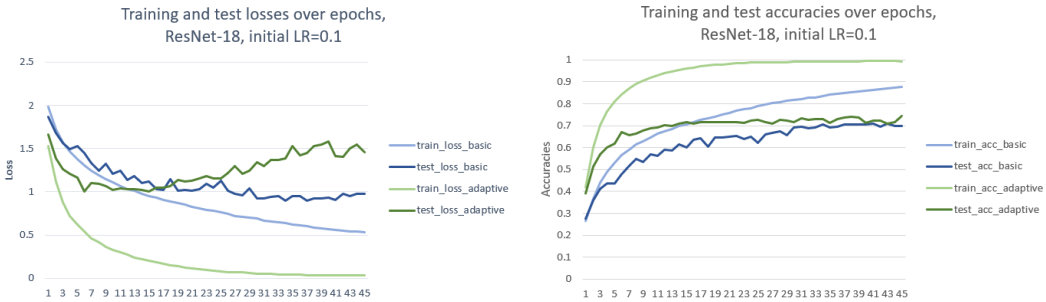

Figure 3: Second-order and basic methods with ResNet, initial learning rate: 0.1

Once again, in the adaptive schedule, all training metrics (training and test loss, training and test accuracies) are optimized faster. Typically, in the first epochs, training loss is twice lower in the adaptive version. The second-order method reaches its plateau sooner as well, after around 15 epochs versus 40 in the basic case, and then starts to overfit. Interestingly, this time the best performance achieved by the adaptive method seems to be slightly better (74.4%) than the best one achieved by the basic method (71%). In the following, we will try several techniques to improve our method's performance. This current experiment with ResNet-18 on Cifar-10 with an initial learning rate of 0.1 will be our reference.

We now explore the variations of the learning rate for both networks:

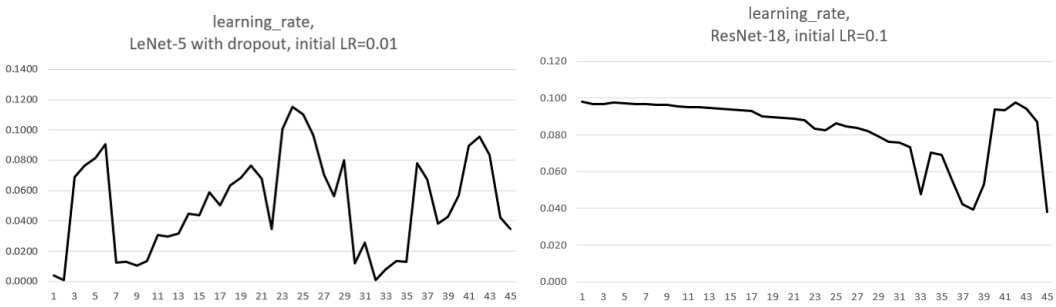

Figure 4: Learning rate variations, LeNet and ResNet, Cifar-10

For LeNet, it seems that the initial learning rate value was a bit too low. The network increases it to boost convergence, but struggles to stabilize to a fixed zone. With ResNet, it seems that a starting learning rate of 0.1 was a bit too high, as we see it decrease its learning rate. Something interesting starts happening here after 30 epochs, as the learning rate starts oscillating. As the loss value stops changing because convergence is reached, it is probable that the denominator in the learning rate updates gets very small.

That leads us to see what happens if we set the initial learning rate at 0.01 for the ResNet model:

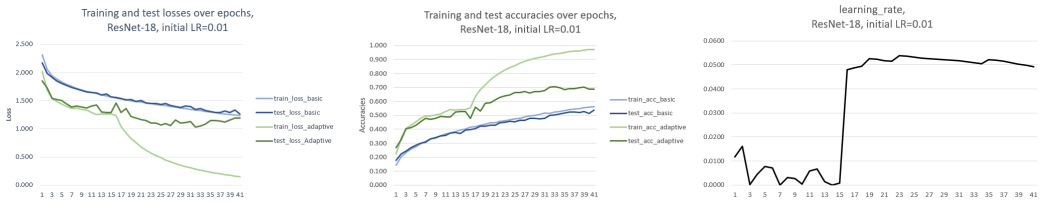

Figure 5: Second-order basic methods with ResNet, initial learning rate: 0.01

The adaptive schedule performs consitently better than the basic one with testing accuracy being 10 to 15 points higher all the way around. This time, the initial learning rate seems to be too small, as the network increases its value until stabilizing it around 0.05, a value around the which it oscillated at the end of the previous training schedule. As training slows down, the network slowly decreases the learning rate value.

First-order method

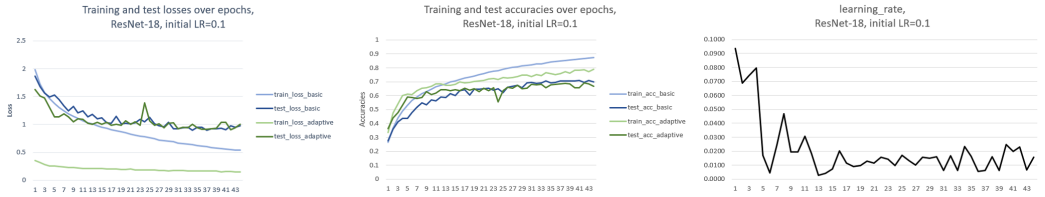

Figure 6: First-order basic methods with ResNet, initial learning rate: 0.1

The first-order method did not bring much of an improvement in comparison to the reference experiment. The tendency to increase overfitting that we noticed with the second-order method is strengthened here, and the learning rate variations are noisier. However, with lower initial learning rates, the first-order method converges sensibly faster than the basic method, but less than the second-order one.

CIFAR-100

In this section, we increment difficulty by solving an image classification problem with 10 times more classes (Cifar-100 [2] has 100 classes). After a line-search, tan ideal learning rate to train ResNet-18 on Cifar-100 seems to be 0.02.

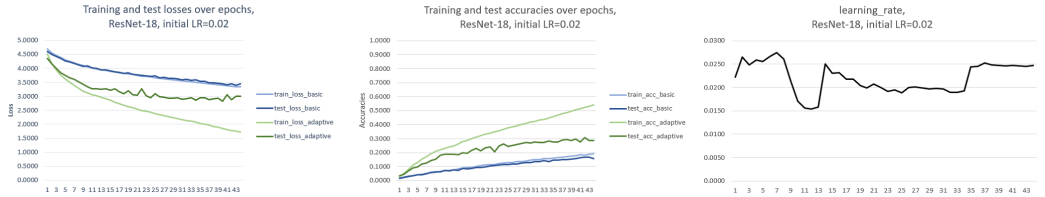

Figure 7: Second-order basic methods with ResNet on Cifar-100, initial learning rate: 0.02

Once again, we see a faster convergence, a better generalization but also a stronger tendency to overfit. The initial learning rate of 0.02 seemed to be a judicious choice as the network barely changes its value over time. We lacked the computational resource to explore training on this dataset for a longer and more relevant period of time.

5 Further exploration

5.1 Learning the learning rate

In the previous experiments, we have seen the learning rate converge to a given value over time (or, at least, stabilize around a certain value). This value seems to depend on both the model and the dataset. Now a question rises: given a dataset and a model, is this the ideal learning rate value to perform gradient descent on this task ?

We started an adaptive learning rate training with the learning rate value that the ResNet model converged to in section 4.2.3 with 0.01 as a starting learning rate. That value is approximatively 0.05

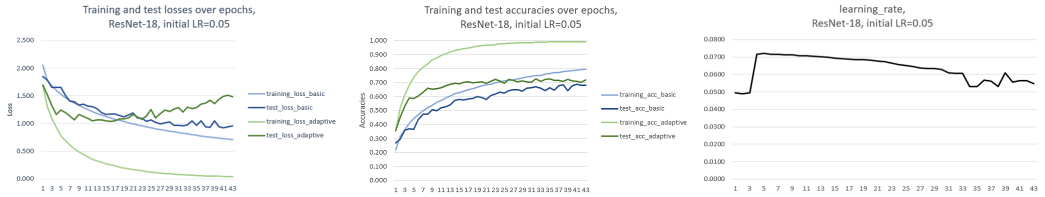

Figure 8: Second-order adaptive learning rate and basic method with ResNet, initial learning rate 0.05

The trends noticed in the previous training are confirmed. This time, the learning rate variations are of much weaker amplitude, as this parameter is already in a ideal zone. The network slightly raises its learning rate to kick off training at first, then lowly decreases it over time, as training converges.

5.2 Momentum

In the same way that momentum reduces noise in a gradient descent for weights, we thought that adding momentum on the learning rate updates could help for both our methods. For the second-order method, the update on η now is:

$$\delta(t+1) = \beta * \delta(t) - 2\epsilon \frac{(f(\eta+\epsilon) - f(\eta-\epsilon))}{f(\eta+2\epsilon) + f(\eta-2\epsilon) - 2f(\eta)} \tag{20}$$

$$\eta(t+1) = \eta(t) + \delta(t+1) \tag{21}$$

Here are the results on our reference experiment with $\beta = 0.9$:

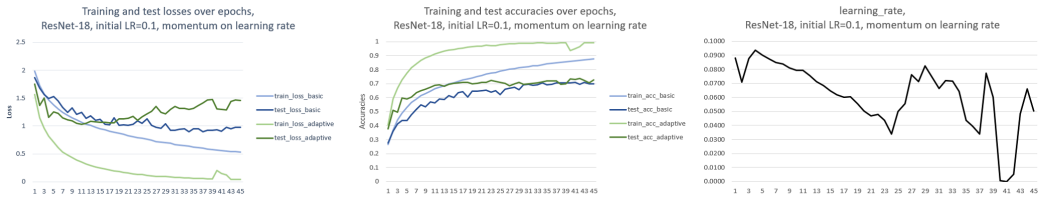

Figure 9: Second-order with momentum and basic methods with ResNet, initial learning rate: 0.1

Such a momentum does not seem to bring any particular improvement here.

5.3 Getting loss values via a validation set

One idea to reduce overfitting tendency in our 2 adaptive methods could be to compute the loss on sample validation batches. This way, the update on the learning rate would guarantee to move in a direction that better reduces the generalization error (and not the training error). The formula for this variant is the same, expect for the use of the f function, that now becomes:

$$f : R^n \to R \tag{22}$$

$$\eta \to L_{\text{random validation batch}}(w(t) - \eta \nabla L_{\text{current training batch}}) \tag{23}$$

We partitioned the original training set in a training and validation set, and applied this technique on the reference experiment:

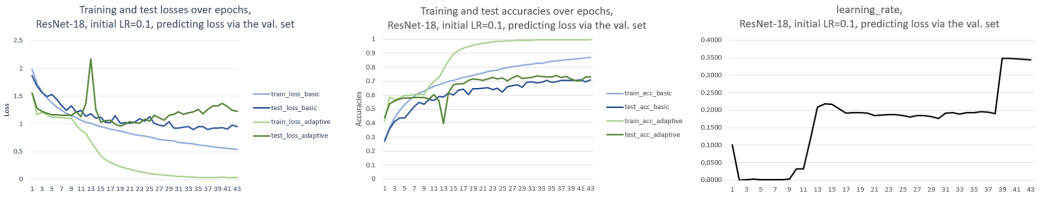

Figure 10: Second-order with val. sampling and basic methods with ResNet, initial learning rate: 0.1

These results suggest that the training loss and training accuracy get optimized slower, but a deeper investigation would be needed to conclude on a reduction of overfitting.

5.4 Comparison with other optimizers

In the last few years, a popular variation of gradient descent named Adam [7] has progressively gained consensus among researchers for its efficiency. Thus, we compared our method with Adam and changed its default learning rate of 10^{-3} to 10^{-1}, our initial value for the adaptive method.

Here is the comparison with our reference experiment, which results were first shown in section 4.2.3:

Figure 11: Second-order and basic method using Adam with ResNet, initial learning rate: 0.1

In the first epochs, our method performs better than an Adam optimization, but the overfitting difference is wider.

6 Limitations

6.1 Choice of step ϵ

In finite differences, the choice of the paramater ϵ is a main issue. Indeed, in any deep learning problem, we can expect the loss function to present noisy oscillations locally. On one hand, with

a too small value of ϵ, we might not capture meaningful variations of the f function. On the other hand, a too large ϵ would make the learning rate oscillate too much, reaching either too high values or even negative values. Both cases can in turn quickly make the loss function diverge. We have seen empirically that when the learning rate gets negative, training will likely not converge.

All our reported experiments were done using an ϵ value of 10^{-5}, which seemed to be a good compromise, and showed stable results for a wide variety of tasks. Tuning ϵ for a given problem overthrows the interest of not having to tune a learning rate anymore. Ideally, a deeper investigation of finite differences should be able to give values of ϵ usable for a large variety of tasks.

6.2 Choice of the initial learning rate

In our algorithm schedule, learning rate variations are automatic, but we have to choose the initial rate value.

This value seems not to matter so much, as with several ranges of initial values, our algorithm still does better than the basic method. Besides, it always adapts its learning rate to reach the same zone, whatever the initial value (in a reasonable range of learning rate values, e.g. 10^{-4} to 1 for Cifar-10 for instance).

6.3 Cost of loss computations

We have shown faster training in terms of number of epochs. However, at each iteration, we have to perform five forward passes and back-propagations instead of one. Thus, computation time is slowed down compared to the fixed learning rate approach.

Assuming that the experimenter does not care about time but about finding the best loss function minimization, it is not such a big problem.

6.4 Overfitting

Our version of gradient descent reduces the training loss much faster than in the fixed learning rate approach. The training loss sometimes gets down to very surprisingly low values, while the test loss does not reduce much more than in the basic model. On the contrary, the test loss usually increases and ends at a higher value than the basic method one. Thus, our model seems to be more likely to overfit. We have tried to add dropout to the ResNet model, and that proved successful in reducing the gap between training and testing performance.

However, in terms of accuracy, the second-order method performs systematically as well as the basic method, if not slightly better. This happens surprisingly even when its test loss is higher.

7 Conclusion

In this paper, we have built a new way to learn the learning rate at each step using finite differences on the loss. We have tested it on a variety of convex and non-convex optimization tasks.

Based on our results, we believe that our method would be able to adapt a good learning rate at every iteration on convex problems. In the case of non-convex problems, we repeatedly observed faster training in the first few epochs. However, our adaptive model seems more inclined to overfit the training data, even though its test accuracy is always comparable to standard SGD performance, if not slightly better. Hence we believe that in neural network architectures, our model can be used initially for pretraining for a few epochs, and then continue with any other standard optimization technique to lead to faster convergence and be computationally more efficient, and perhaps reach a new highest accuracy on the given problem.

Moreover, the learning rate that our algorithm converges to suggests an ideal learning rate for the given training task. One could use our method to tune the learning rate of a standard neural network (using Adam for instance), giving a more precise value than with line-search or random search.

References

[1] The boston housing dataset. `https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html`.

[2] Cifar-10 and cifar-100 datasets. `https://www.cs.toronto.edu/~kriz/cifar.html`.

[3] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[5] Geoffrey E Hinton. Training products of experts by minimizing contrastive divergence. *Training*, 14(8), 2006.

[6] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[7] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[9] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[10] James Martens. Deep learning via hessian-free optimization. In *ICML*, volume 27, pages 735–742, 2010.

[11] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[12] Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. In *International Conference on Machine Learning*, pages 343–351, 2013.

[13] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[14] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.