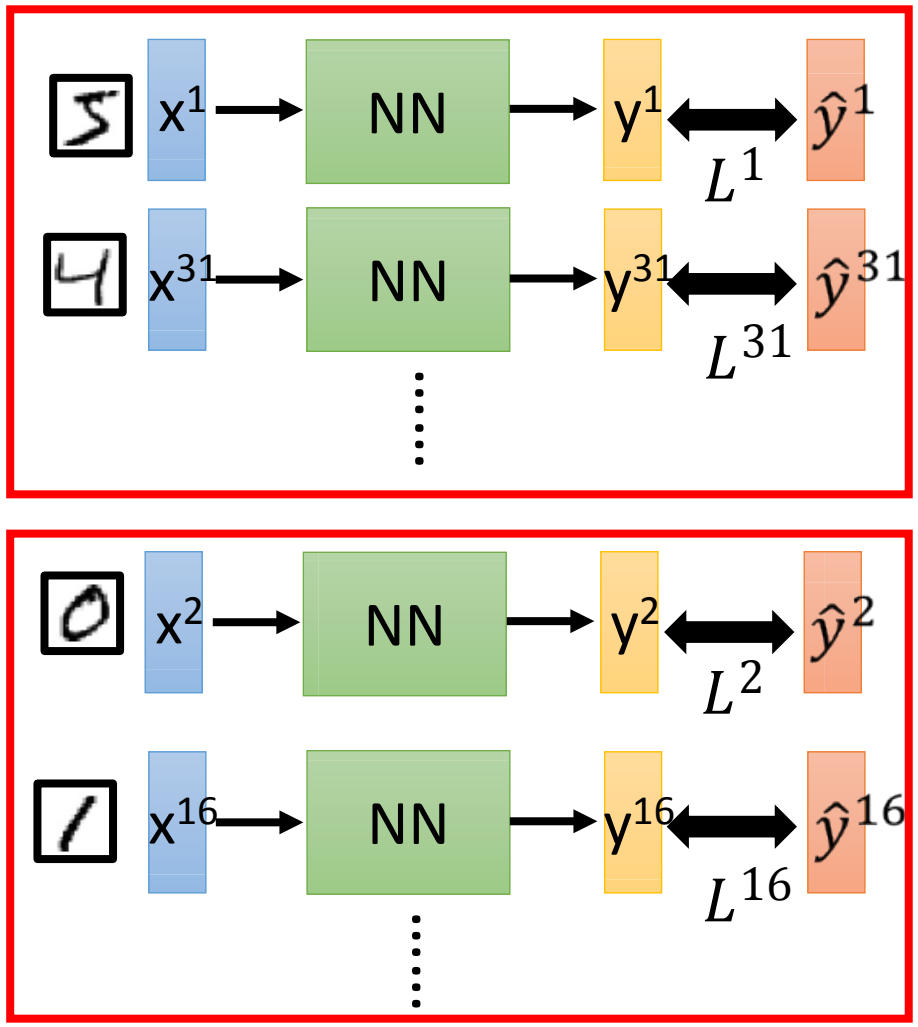
CSE463: Neural Networks

Back to Optimization

by:
Hossam Abd El Munim
Computer & Systems Engineering Dept.,
Ain Shams University,
1 El-Sarayat Street, Abbassia, Cairo 11517

Mini-batch



- Randomly initialize θ^0
- Pick the 1st batch

$$C = L^1 + L^{31} + \cdots$$

$$\theta^1 \leftarrow \theta^0 - \eta \nabla C(\theta^0)$$

- Pick the 2nd batch

$$C = L^2 + L^{16} + \cdots$$

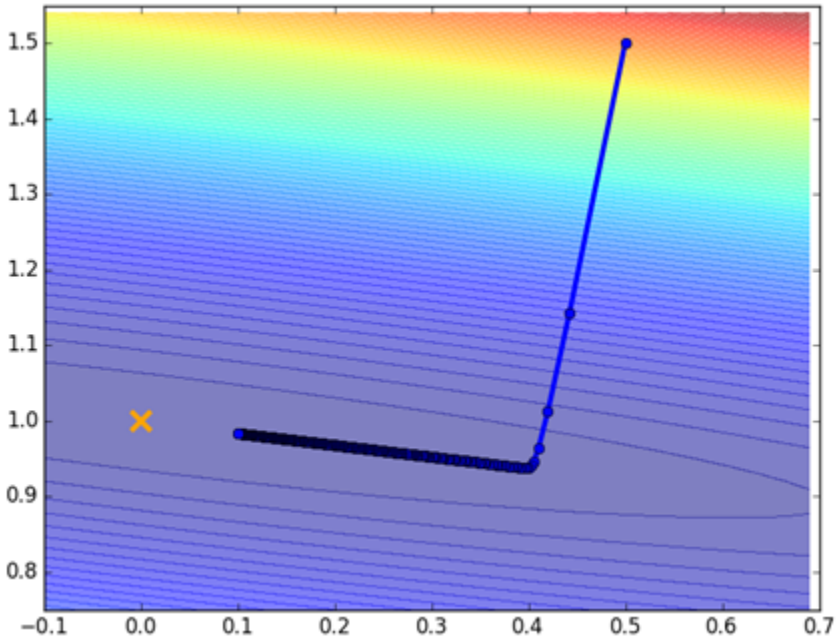
$$\theta^2 \leftarrow \theta^1 - \eta \nabla C(\theta^1)$$

$\vdots$

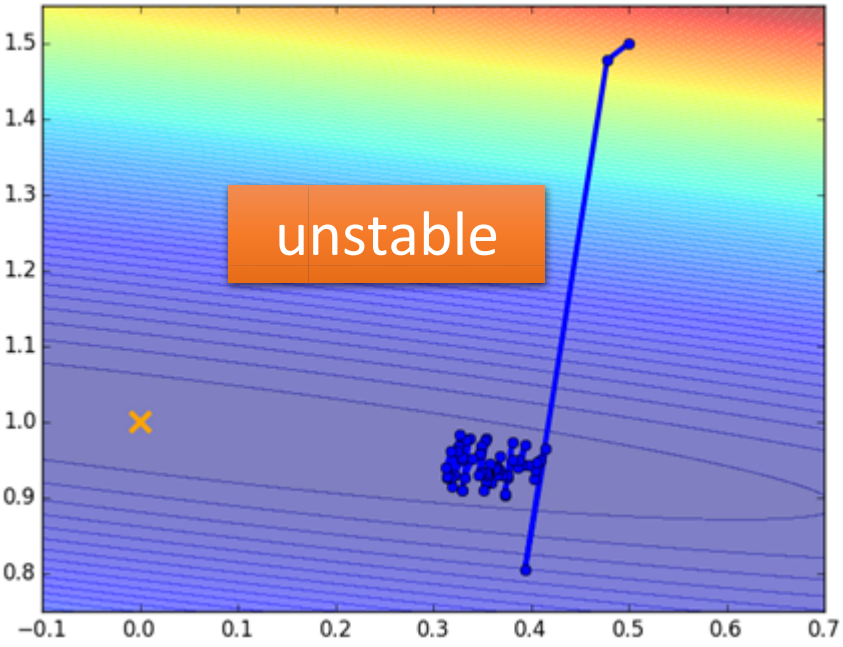
C is different each time when we update parameters!

Mini-batch (Stochastic Gradient Descent (SGD))

Original Gradient Descent



With Mini-batch

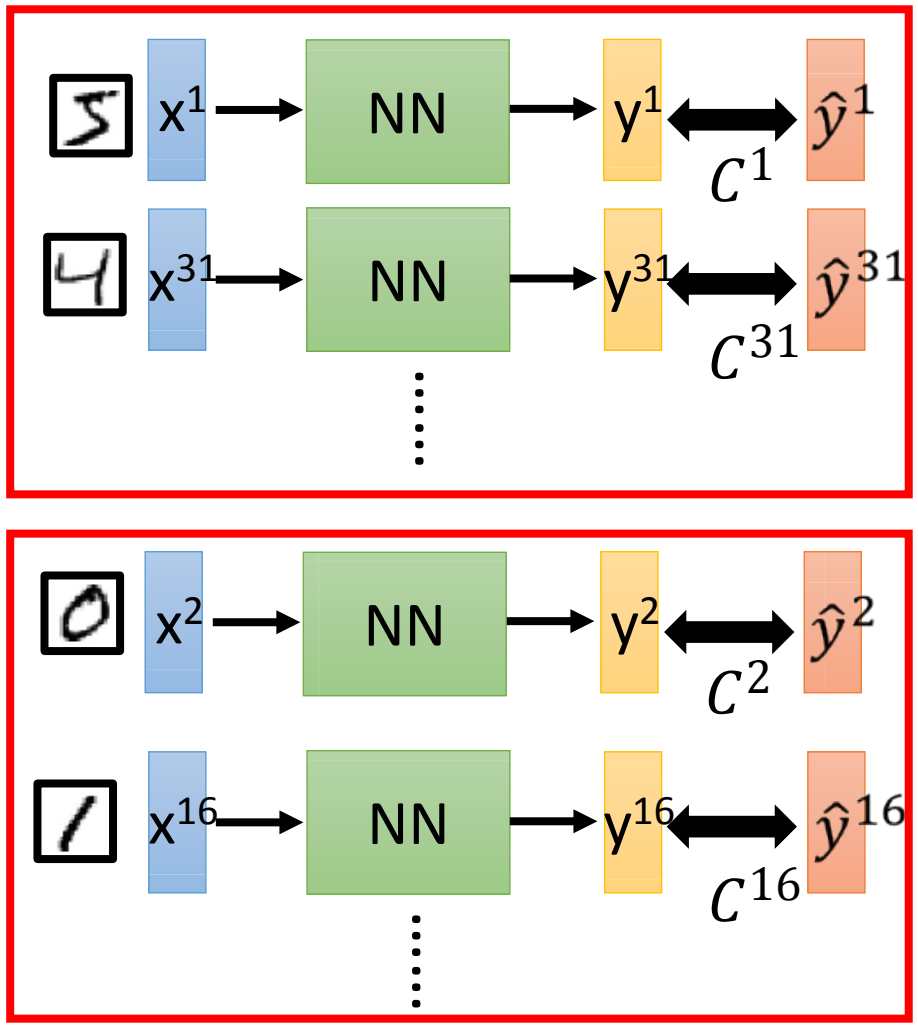


The colors represent the total C on all training data.

Mini-batch

Faster

Better!



- Randomly initialize θ^0
- Pick the 1st batch

$$C = C^1 + C^{31} + \cdots$$

$$\theta^1 \leftarrow \theta^0 - \eta \nabla C(\theta^0)$$

- Pick the 2nd batch

$$C = C^2 + C^{16} + \cdots$$

$$\theta^2 \leftarrow \theta^1 - \eta \nabla C(\theta^1)$$

⋮

- Until all mini-batches have been picked

one epoch

Repeat the above process

Backpropagation

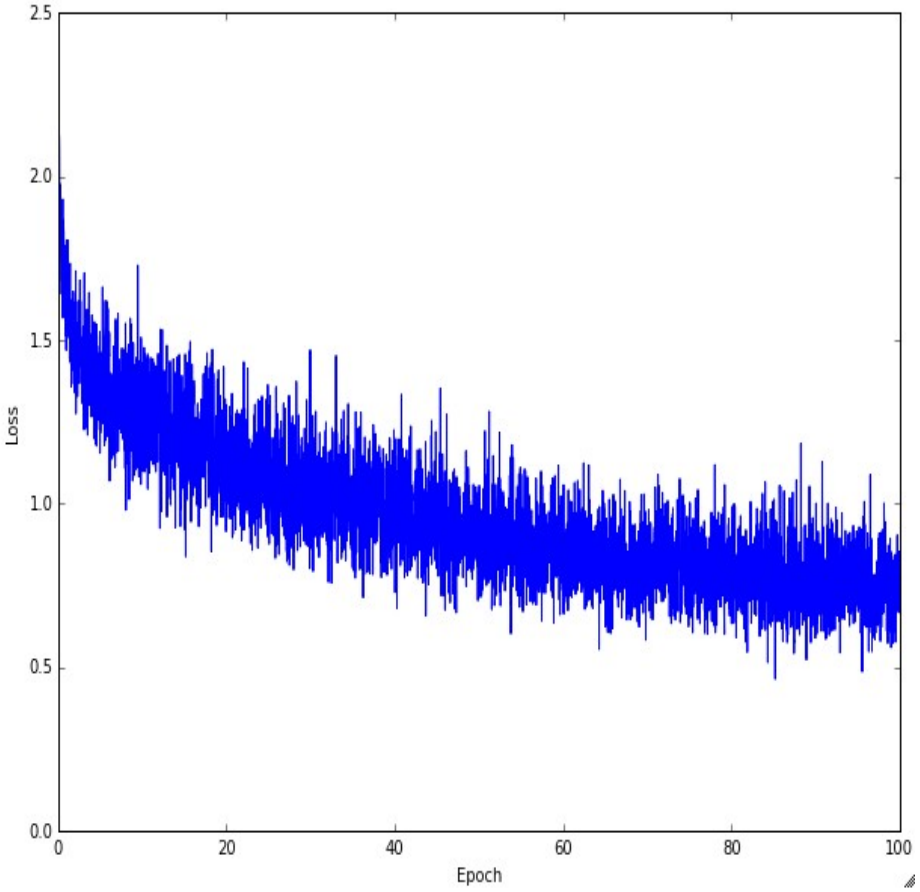
- A network can have millions of parameters.
  - Backpropagation is the way to compute the gradients efficiently (not today)
  - Ref: http://speech.ee.ntu.edu.tw/~tlkagk/courses/MLDS_2015_2/Lecture/DNN%20backprop.ecm.mp4/index.html
- Many toolkits can compute the gradients automatically

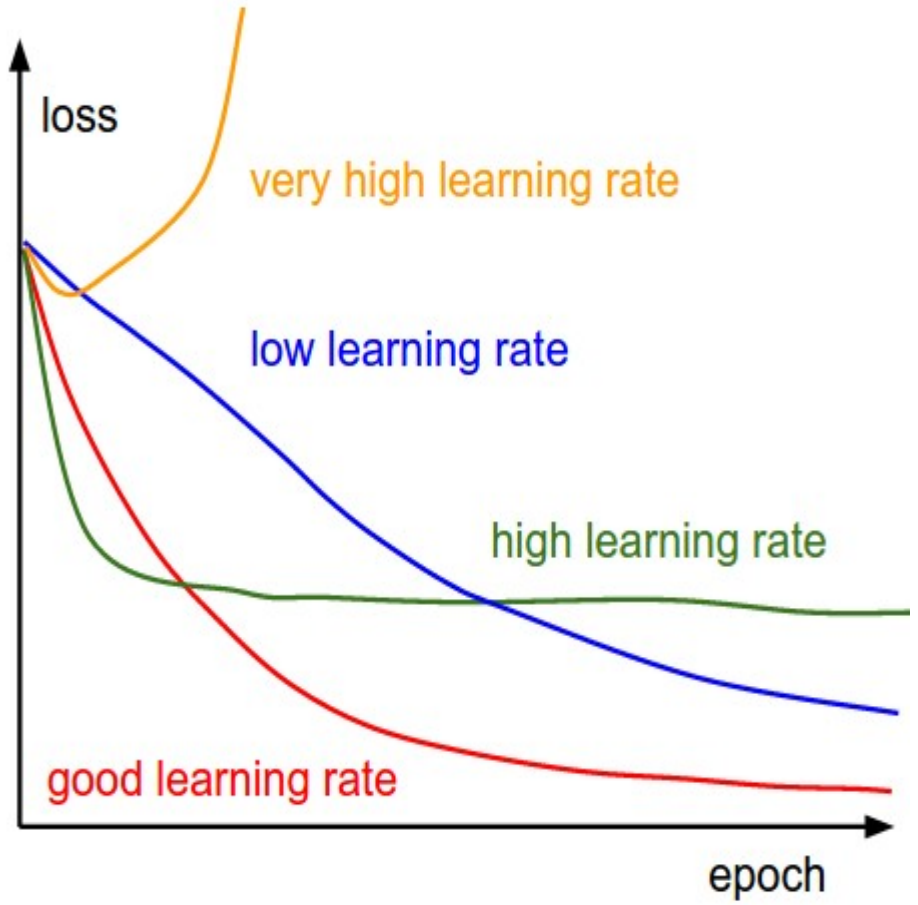
theano

TensorFlow

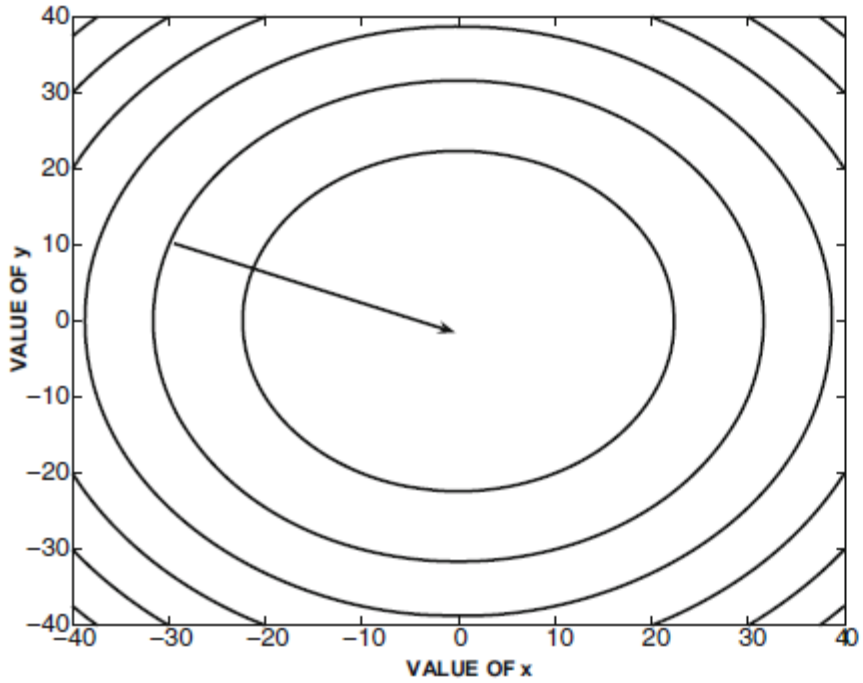
Ref: http://speech.ee.ntu.edu.tw/~tlkagk/courses/MLDS_2015_2/Lecture/Theano%20DNN.ecm.mp4/index.html

The effects of step size (or “learning rate”)

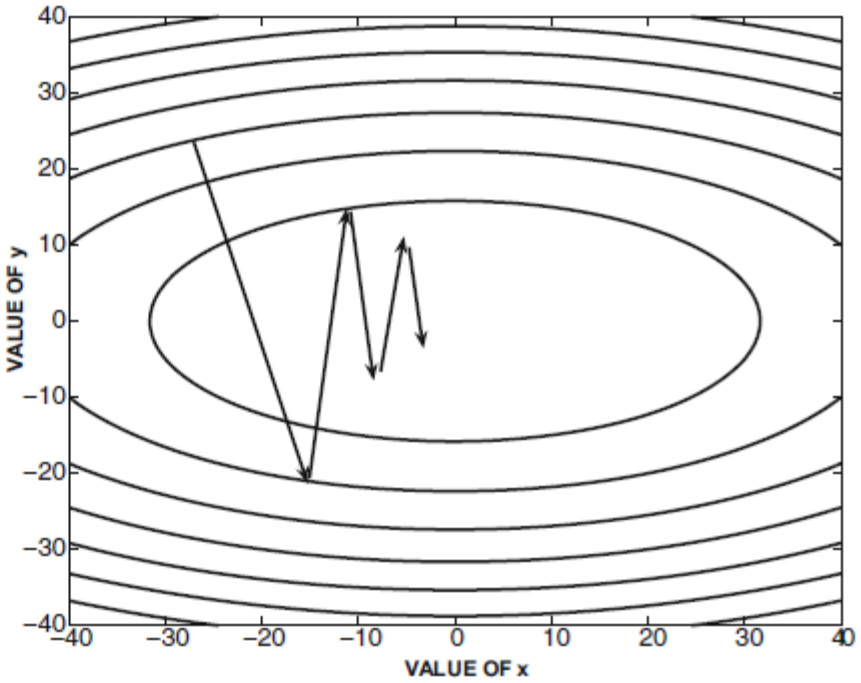




Geometric Understanding of the Effect of Gradient Ratios



(a) Loss function is circular bowl
$L = x^2 + y^2$



(b) Loss function is elliptical bowl
$L = x^2 + 4y^2$

Vanishing gradients have the same effect.

Learning Rate Decay

$$\alpha_t = \alpha_0 \exp(-k \cdot t) \quad \text{[Exponential Decay]}$$

$$\alpha_t = \frac{\alpha_0}{1 + k \cdot t} \quad \text{[Inverse Decay]}$$

The parameter k controls the rate of the decay. Another approach is to use step decay in which the learning rate is reduced by a particular factor every few epochs. For example, the learning rate might be multiplied by 0.5 every 5 epochs. A common approach is to track the loss on a held-out portion of the training data set, and reduce the learning rate whenever this loss stops improving. In some cases, the analyst might even babysit the learning process, and use an implementation in which the learning rate can be changed manually depending on the progress. This type of approach can be used with simple implementations of gradient descent, although it does not address many of the other problematic issues.

Momentum-Based Learning

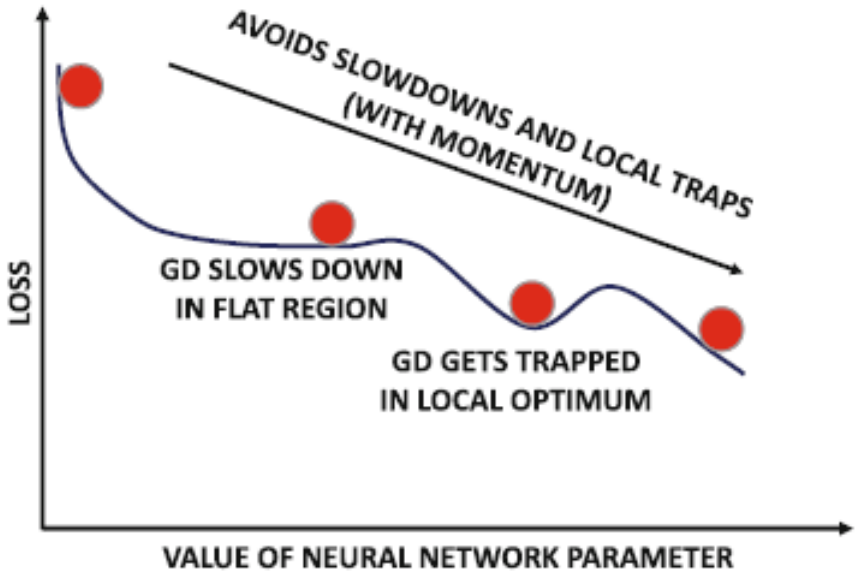


Figure 3.12: Effect of momentum in navigating complex loss surfaces. The annotation "GD" indicates pure gradient descent without momentum. Momentum helps the optimization process retain speed in flat regions of the loss surface and avoid local optima.

Momentum-Based Learning (Cont.…)

Here, α is the learning rate. In momentum-based descent, the vector $\overline{V}$ is modified with exponential smoothing, where $\beta \in (0, 1)$ is a smoothing parameter:

$$\overline{V} \Leftarrow \beta \overline{V} - \alpha \frac{\partial L}{\partial \overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

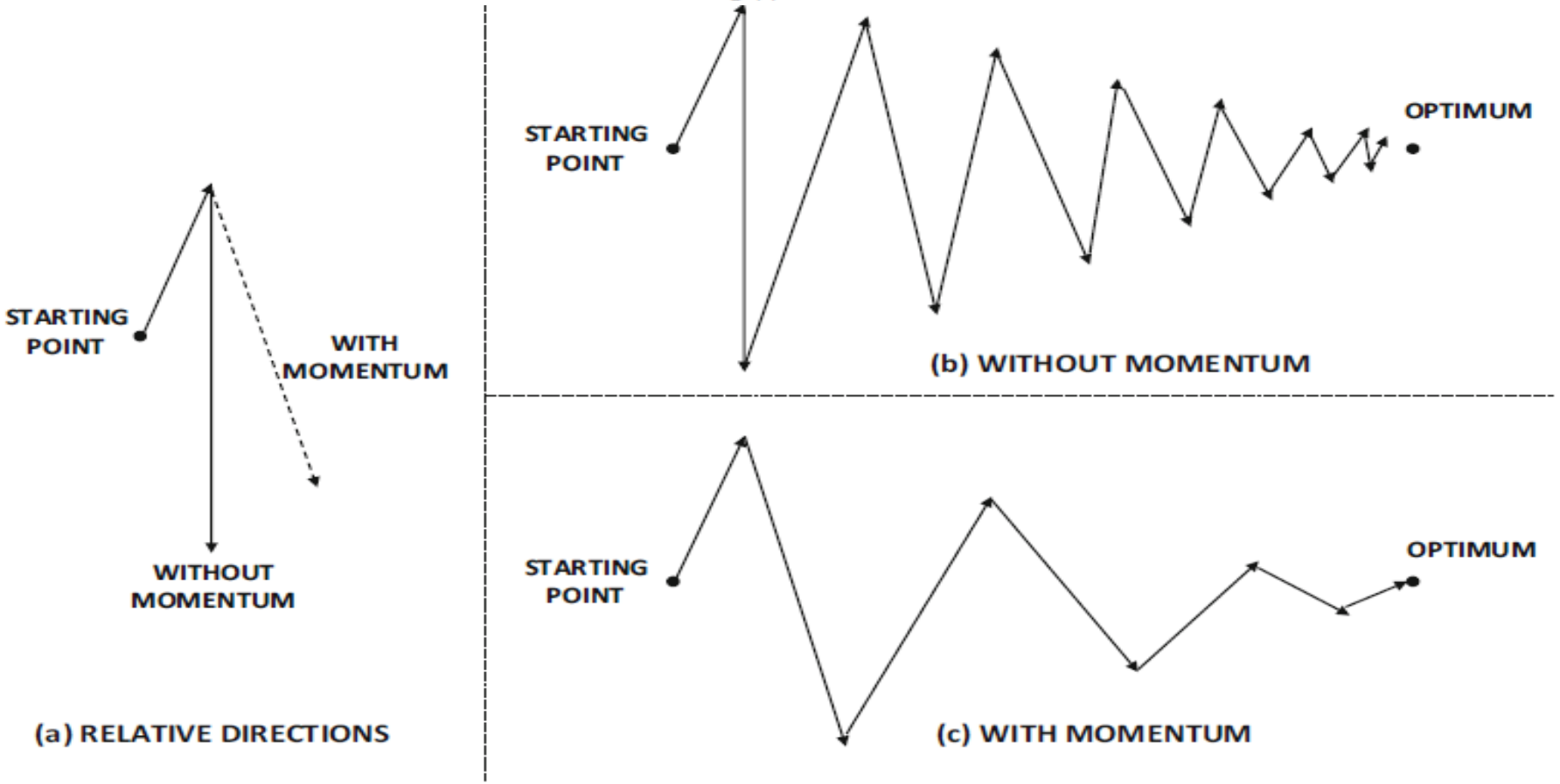


Figure 3.11: Effect of momentum in smoothing zigzag updates

Momentum-Based Learning (Cont.…)

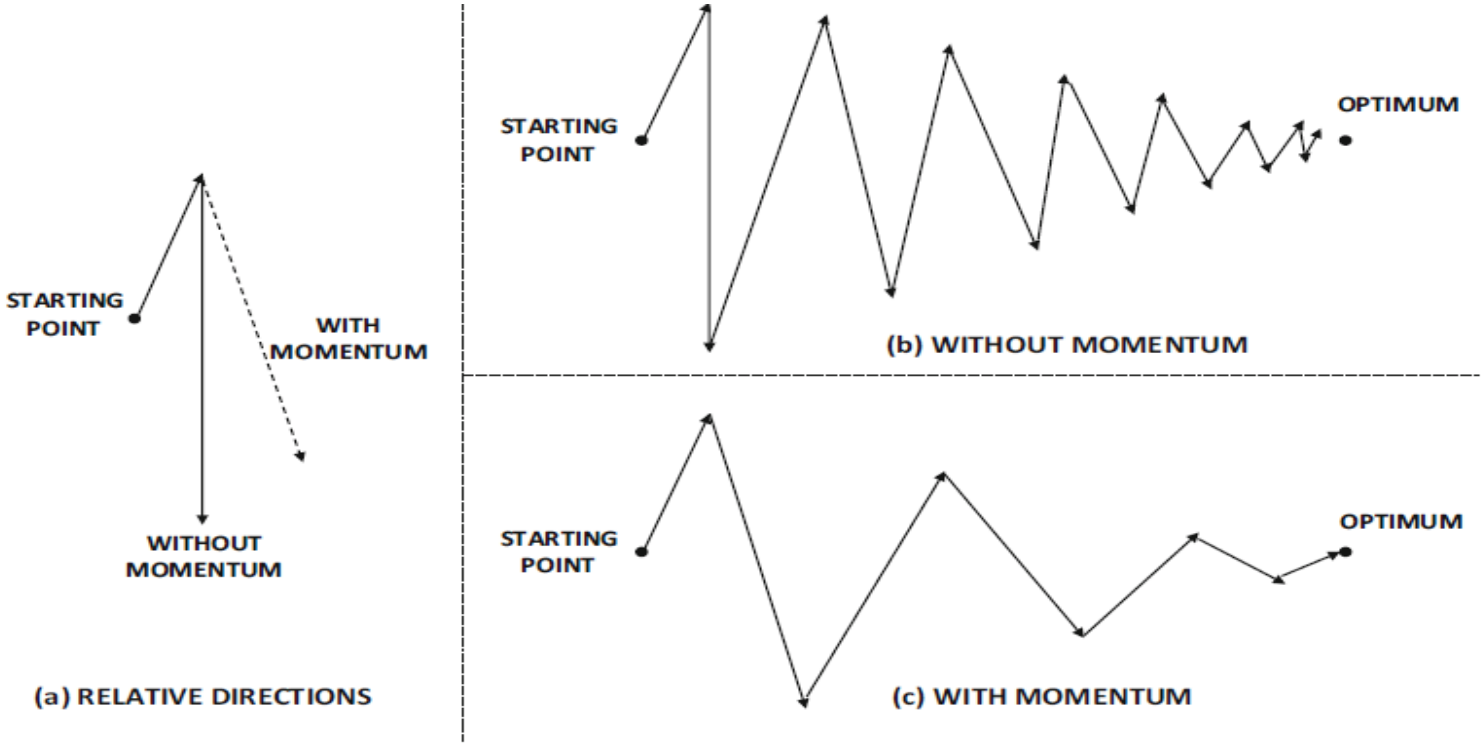


Figure 3.11: Effect of momentum in smoothing zigzag updates

Figure 3.11. It is evident from Figure 3.11(a) that momentum increases the relative component of the gradient in the correct direction. The corresponding effects on the updates are illustrated in Figure 3.11(b) and (c). It is evident that momentum-based updates can reach the optimal solution in fewer updates.

Nesterov Momentum

Let us denote the loss function by $L(\overline{W})$ at the current solution $\overline{W}$. In this case, it is important to explicitly denote the argument of the loss function because of the way in which the gradient is computed. Therefore, the update may be computed as follows:

$$\overline{V} \Leftarrow \beta \overline{V} - \alpha \frac{\partial L(\overline{W} + \beta \overline{V})}{\partial \overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

Note that the *only* difference from the standard momentum method is in terms of *where* the gradient is computed. Using the value of the gradient a little further along the previous update can lead to faster convergence.

Parameter-Specific Learning Rates AdaGrad

Let A_i be the aggregate value for the ith parameter. Therefore, in each iteration, the following update is performed:

$$A_i \Leftarrow A_i + \left(\frac{\partial L}{\partial w_i}\right)^2 \quad \forall i \tag{3.40}$$

The update for the ith parameter w_i is as follows:

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}} \left(\frac{\partial L}{\partial w_i}\right); \quad \forall i$$

If desired, one can use $\sqrt{A_i + \epsilon}$ in the denominator instead of $\sqrt{A_i}$ to avoid ill-conditioning. Here, ϵ is a small positive value such as 10^{-8}.

- **Progress along "steep" directions is damped;**
- **progress along "flat" directions is accelerated;**
- **If it does not converge, step size decays to zero..**

Parameter-Specific Learning Rates RMSProp: "Leaky AdaGrad"

$$A_i \Leftarrow \rho A_i + (1-\rho)\left(\frac{\partial L}{\partial w_i}\right)^2 \quad \forall i \tag{3.41}$$

The square-root of this value for each parameter is used to normalize its gradient. Then, the following update is used for (global) learning rate α:

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}}\left(\frac{\partial L}{\partial w_i}\right); \quad \forall i$$

RMSProp with Nesterov Momentum

RMSProp can also be combined with Nesterov momentum. Let A_i be the squared aggregate of the ith weight. In such cases, we introduce the additional parameter $\beta \in (0, 1)$ and use the following updates:

$$v_i \Leftarrow \beta v_i - \frac{\alpha}{\sqrt{A_i}} \left(\frac{\partial L(\overline{W} + \beta \overline{V})}{\partial w_i} \right); \quad w_i \Leftarrow w_i + v_i \quad \forall i$$

Note that the partial derivative of the loss function is computed at a shifted point, as is common in the Nesterov method. The weight $\overline{W}$ is shifted with $\beta \overline{V}$ while computing the partial derivative with respect to the loss function. The maintenance of A_i is done using the shifted gradients as well:

$$A_i \Leftarrow \rho A_i + (1 - \rho) \left(\frac{\partial L(\overline{W} + \beta \overline{V})}{\partial w_i} \right)^2 \quad \forall i \tag{3.42}$$

Although this approach benefits from adding momentum to RMSProp, it does not correct for the initialization bias.

AdaDelta

The AdaDelta algorithm [553] uses a similar update as RMSProp, except that it eliminates the need for a global learning parameter by computing it as a function of incremental updates in previous iterations. Consider the update of RMSProp, which is repeated below:

$$w_i \Leftarrow w_i - \underbrace{\frac{\alpha}{\sqrt{A_i}}\left(\frac{\partial L}{\partial w_i}\right)}_{\Delta w_i}; \quad \forall i$$

We will show how α is replaced with a value that depends on the previous incremental updates. In each update, the value of Δw_i is the increment in the value of w_i. As with the exponentially smoothed gradients A_i, we keep an exponentially smoothed value δ_i of the values of Δw_i in previous iterations with the same decay parameter ρ:

$$\delta_i \Leftarrow \rho\delta_i + (1-\rho)(\Delta w_i)^2 \quad \forall i \tag{3.43}$$

For a given iteration, the value of δ_i can be computed using only the iterations before it because the value of Δw_i is not yet available. On the other hand, A_i can be computed using the partial derivative in the current iteration as well. This is a subtle difference between how A_i and δ_i are computed. This results in the following AdaDelta update:

$$w_i \Leftarrow w_i - \underbrace{\sqrt{\frac{\delta_i}{A_i}}\left(\frac{\partial L}{\partial w_i}\right)}_{\Delta w_i}; \quad \forall i$$

Adam (1)

The Adam algorithm uses a similar "signal-to-noise" normalization as AdaGrad and RMSProp; however, it also exponentially smooths the first-order gradient in order to incorporate momentum into the update. It also directly addresses the bias inherent in exponential smoothing when the running estimate of a smoothed value is unrealistically initialized to 0.

As in the case of RMSProp, let A_i be the exponentially averaged value of the ith parameter w_i. This value is updated in the same way as RMSProp with the decay parameter $\rho \in (0, 1)$:

$$A_i \Leftarrow \rho A_i + (1 - \rho) \left(\frac{\partial L}{\partial w_i} \right)^2 \quad \forall i \tag{3.44}$$

At the same time, an exponentially smoothed value of the gradient is maintained for which the ith component is denoted by F_i. This smoothing is performed with a different decay parameter ρ_f:

$$F_i \Leftarrow \rho_f F_i + (1 - \rho_f) \left(\frac{\partial L}{\partial w_i} \right) \quad \forall i \tag{3.45}$$

Adam (2)

This type of exponentially smoothing of the gradient with ρ_f is a variation of the momentum method discussed in Section 3.5.2 (which is parameterized by a friction parameter β instead of ρ_f). Then, the following update is used at learning rate α_t in the tth iteration:

$$w_i \Leftarrow w_i - \frac{\alpha_t}{\sqrt{A_i}} F_i; \quad \forall i$$

There are two key differences from the RMSProp algorithm. First, the gradient is replaced with its exponentially smoothed value in order to incorporate momentum. Second, the learning rate α_t now depends on the iteration index t, and is defined as follows:

$$\alpha_t = \alpha \underbrace{\left(\frac{\sqrt{1-\rho^t}}{1-\rho_f^t} \right)}_{\text{Adjust Bias}} \tag{3.46}$$

Adam (3)

Technically, the adjustment to the learning rate is actually a bias correction factor that is applied to account for the unrealistic initialization of the two exponential smoothing mechanisms, and it is particularly important in early iterations. Both F_i and A_i are initialized to 0, which causes bias in early iterations. The two quantities are affected differently by the bias, which accounts for the ratio in Equation 3.46. It is noteworthy that each of ρ^t and ρ_f^t converge to 0 for large t because $\rho, \rho_f \in (0, 1)$. As a result, the initialization bias correction factor of Equation 3.46 converges to 1, and α_t converges to α. The default suggested values of ρ_f and ρ are 0.9 and 0.999, respectively, according to the original Adam paper [241]. Refer to [241] for details of other criteria (such as parameter sparsity) used for selecting ρ and ρ_f. Like other methods, Adam uses $\sqrt{A_i + \epsilon}$ (instead of $\sqrt{A_i}$) in the denominator of the update for better conditioning. The Adam algorithm is extremely popular because it incorporates most of the advantages of other algorithms, and often performs competitively with respect to the best of the other methods [241].

Second Order Derivatives (1)

A number of methods have been proposed in recent years for using second-order derivatives for optimization. Such methods can partially alleviate some of the problems caused by curvature of the loss function.

Consider the parameter vector $\overline{W} = (w_1 \ldots w_d)^T$, which is expressed[3] as a column vector. The second-order derivatives of the loss function $L(\overline{W})$ are of the following form:

$$H_{ij} = \frac{\partial^2 L(\overline{W})}{\partial w_i \partial w_j}$$

Note that the partial derivatives use all pairwise parameters in the denominator. Therefore, for a neural network with d parameters, we have a $d \times d$ *Hessian matrix* H, for which the (i, j)th entry is H_{ij}. The second-order derivatives of the loss function can be computed with backpropagation [315], although this is rarely done in practice. The Hessian can be viewed as the Jacobian of the gradient.

One can write a quadratic approximation of the loss function in the vicinity of parameter vector $\overline{W}_0$ by using the following Taylor expansion:

$$L(\overline{W}) \approx L(\overline{W}_0) + (\overline{W} - \overline{W}_0)^T [\nabla L(\overline{W}_0)] + \frac{1}{2}(\overline{W} - \overline{W}_0)^T H (\overline{W} - \overline{W}_0) \tag{3.47}$$

Second Order Derivatives (2)

Note that the Hessian H is computed at $\overline{W}_0$. Here, the parameter vectors $\overline{W}$ and $\overline{W}_0$ are d-dimensional column vectors, as is the gradient of the loss function. This is a quadratic approximation, and one can simply set the gradient to 0, which results in the following optimality condition for the quadratic approximation:

$$\nabla L(\overline{W}) = 0 \quad \text{[Gradient of Loss Function]}$$
$$\nabla L(\overline{W}_0) + H(\overline{W} - \overline{W}_0) = 0 \quad \text{[Gradient of Taylor approximation]}$$

One can rearrange the above optimality condition to obtain the following Newton update:

$$\overline{W}^* \Leftarrow \overline{W}_0 - H^{-1}[\nabla L(\overline{W}_0)] \tag{3.48}$$

Second Order Derivatives (3)

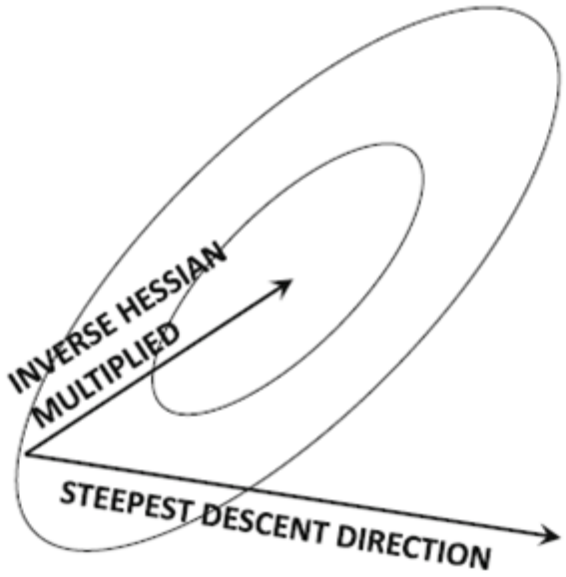


Figure 3.14: The effect of pre-multiplication of steepest-descent direction with the inverse Hessian

Second Order Derivatives (4)

The main difference of Equation 3.48 from the update of steepest-gradient descent is pre-multiplication of the steepest direction (which is $[\nabla L(\overline{W}_0)]$) with the inverse of the Hessian. This multiplication with the inverse Hessian plays a key role in changing the direction of the steepest-gradient descent, so that one can take larger steps in that direction (resulting in better improvement of the objective function) even if the *instantaneous* rate of change in that direction is not as large as the steepest-descent direction. This is because the Hessian encodes how fast the gradient is changing in each direction. Changing gradients are bad for larger updates because one might inadvertently worsen the objective function, if the signs of many components of the gradient change during the step. It is profitable to move in directions where the ratio of the gradient to the rate of change of the gradient is large, so that one can take larger steps without causing harm to the optimization. Pre-multiplication with the inverse of the Hessian achieves this goal. The effect of the pre-multiplication of the steepest-descent direction with the inverse Hessian is shown in Figure 3.14. It is helpful to reconcile this figure with the example of the quadratic bowl in Figure 3.9. In a sense, pre-multiplication with the inverse Hessian biases the learning steps towards low-curvature directions. In one dimension, the Newton step is simply the ratio of the first derivative (rate of change) to the second derivative (curvature). In multiple dimensions, the low-curvature directions tend to win out because of multiplication by the inverse Hessian.

What makes it in-practical? What is the solution?

In most large-scale neural network settings, the Hessian is too large to store or compute explicitly. It is not uncommon to have neural networks with millions of parameters. Trying to compute the inverse of a $10^6 \times 10^6$ Hessian matrix is impractical with the computational power available today. In fact, it is difficult to even compute the Hessian, let alone invert it! Therefore, many approximations and variations of the Newton method have been developed. Examples of such methods include *Hessian-free optimization* [41, 189, 313, 314] (or method of *conjugate gradients*) and quasi-Newton methods that approximate the Hessian. The basic goal of these methods to make second-order updates without exactly computing the Hessian.