Gradient Descent for DL

Alvaro Soto

Computer Science Department, PUC

First order gradient descent optimization

Goal: minimization of a loss function using an iterative gradient descent (steepest descent) approach or SGD.

- Loss function:

$$L(W) = \sum_n \mathcal{L}(f(x_n), y_n; W) + \alpha\, \Omega(W)$$

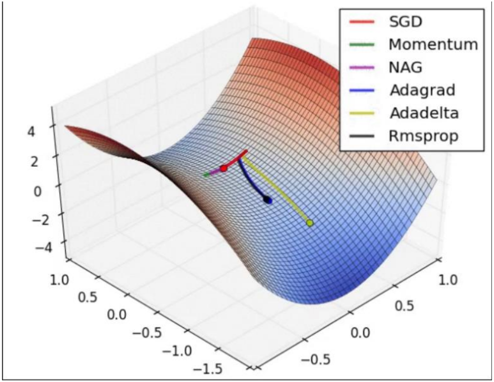
- Weight update given by a step in the direction against the gradient of the loss:

$$w_i^{new} = w_i^{old} - \eta \frac{\partial L}{\partial w_i}$$

We can apply this iterative scheme using a batch (incremental) SGD approach.

Gradient descent in the context of DL: Not an easy road!

Gradient descent problems



SGD + History

- SGD can make erratic updates on non-smooth loss functions.
- SGD will frequently follow the wrong gradient.

We can use the previous gradient to smooth updates, so we can avoid to follow spurious gradient directions.

$$w_i^t = w_i^{t-1} - \left\{ \eta \frac{\partial L(W^t)}{\partial w_i} + \tau \frac{\partial L(\color{red}{W^{t-1}})}{\partial w_i} \right\}$$

$$w_i^t = w_i^{t-1} - \{ \eta \, G^t + \tau \, \color{red}{G^{t-1}} \}$$

SGD + Momentum

Actually, we can smooth accumulating all previous gradients using a recursive or moving average scheme: Momentum Method.

$$w_i^t = w_i^{t-1} - \alpha \, v^t$$

$$v^t = \frac{\partial L(W^t)}{\partial w_i} + \tau \, v^{t-1}$$

- We provide the gradient descent with a short-term memory.
- We accumulate history where recent gradients receive more attention.
- Intuition from physics: a moving ball acquires momentum, at which point it becomes less sensitive to a perturbation (spurious gradient).
- With momentum update, the parameter vector will build up velocity in any direction that has consistent gradient updates.

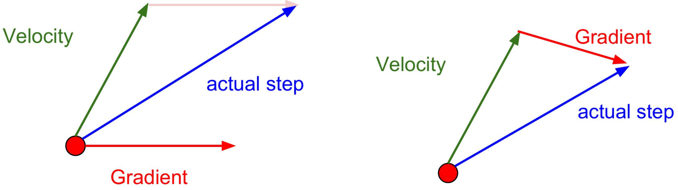
Nesterov Accelerated Gradient (NAG) (Nesterov, 1983; Sutskever et al., 2013)

IDEA: We can use momentum to look forward the evolution of the gradient optimization. Then use the gradient to correct or reinforce this direction.

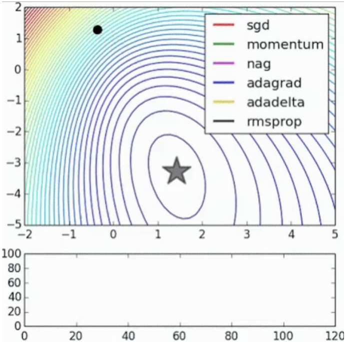
- First, follows the direction given by the momentum.
- Then, calculate the gradient in the landing position and use it to make a correction.

$$w_i^{t-} = w_i^{t-1} - \tau v_{t-1}$$

$$w_i^{t} = w_i^{t-1} - \eta \frac{\partial L(W^{t-})}{\partial w_i}$$

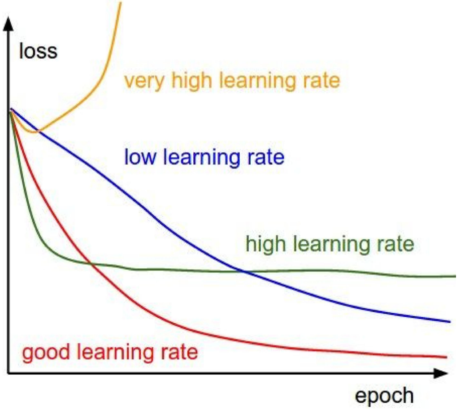


Nesterov Accelerated Gradient (NAG) (Nesterov, 1983; Sutskever et al., 2013)



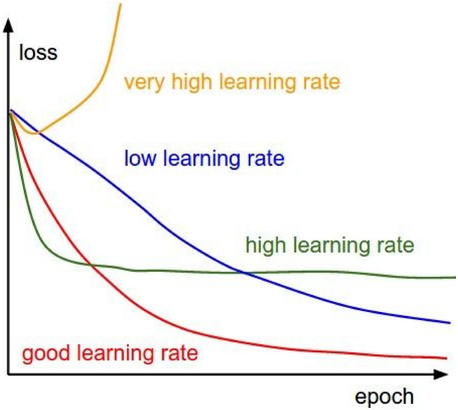
Note: observe that some methods overshoot the goal, any hypothesis?

Learning Rate is Key



- If learning rate is too large, cost might not decrease after each update (overshoot).
- If learning rate is too small, training would be too slow.

Learning Rate is Key



Can we use a different learning rate to update each parameter?

Adaptive Gradient Methods

Adaptive Gradient: AdaGrad (Duchi et al., 2010)

Main Idea: Each weight has its owns learning rate.

AdaGrad: Scale each gradient based on its history. Specifically, historic sum of squares in each dimension (each parameter).

$$w_i^t = w_i^{t-1} - \eta_{w_i} \frac{\partial L(W^t)}{\partial w_i}$$

$$\eta_{w_i} = \frac{\eta}{\sqrt{\sum_{i=0}^{t} \{G_i^t\}^2}}$$

- η_{w_i}: constant divided by sum of square of previous derivatives.
    - Numerator η, global learning rate shared by all parameters.
    - Denominator computes the ℓ_2-norm of all previous gradients on a per-parameter basis.

Adaptive Gradient: AdaGrad (Duchi et al., 2010)

Main Idea: Each weight has its owns learning rate.

AdaGrad: Scale each gradient based on its history. Specifically, historic sum of squares in each dimension (each parameter).

$$w_i^t = w_i^{t-1} - \eta_{w_i} \frac{\partial L(W^t)}{\partial w_i}$$

$$\eta_{w_i} = \frac{\eta}{\sqrt{\sum_{i=0}^{t} \{G_i^t\}^2}}$$

- Good Effect: AdaGrad scales updates, i.e., magnitudes of gradients are mostly factored out. In other words, smaller derivative implies larger learning rate, and vice versa , why?
- No so good: Learning rate keeps decreasing for all parameters. Actually, AdaGrad lowers the update size very aggressively, why?

RMSprop (Root Mean Square propagation) (Tieleman and Hinton, 2012)

- Idea: limit sum of squared gradients to a restricted window of past gradients.
- Actually, we can use a moving average, so we scale by decaying average of squared gradient (instead of sum of squared gradients as in AdaGrad).

$$w_i^t = w_i^{t-1} - \eta_{w_i} \frac{\partial L(W^t)}{\partial w_i}$$

$$\eta_{w_i} = \frac{\eta}{\sqrt{r_t}}$$

$$r_t = (1-\gamma)\{G_i^t\}^2 + \gamma\, r_{t-1}$$

- A similar idea is used by AdaDelta (Zeiler, 2012).

RMSprop (Root Mean Square propagation) (Tieleman and Hinton, 2012)

$$w_i^t = w_i^{t-1} - \eta_{w_i} \frac{\partial L(W^t)}{\partial w_i}$$

$$\eta_{w_i} = \frac{\eta}{\sqrt{r_t}}$$

$$r_t = (1-\gamma)\{G_i^t\}^2 + \gamma\, r_{t-1}$$

- RMSprop scales each gradient by a moving average of its squared previous values.
- RMSprop is one of the most popular optimizers for DL.
- In practice works well.

Adam (Adaptive moments) (Kingma and Ba, 2014)

Idea: adaptively adjusts learning rate so that parameters that have changed infrequently based on historical gradients are updated more quickly than parameters that have changed frequently.

- Adam uses estimations of first and second moments of gradient.
- n-th moment m_n of a random variable x is given by: $m_n = E(x^n)$.
- First moment is the mean: r_t. Second moment is the uncentered variance: v_t.

$$r_t = (1-\gamma_1)G_i^t + \gamma_1\, r_{t-1} \qquad v_t = (1-\gamma_2)\{G_i^t\}^2 + \gamma_2\, v_{t-1}$$

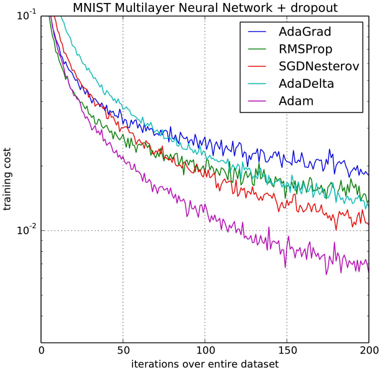
Estimations of moments using moving averages are biased, so we need to compensate dividing by the bias:

$$r_t = \frac{(1-\gamma_1)G_i^t + \gamma_1\, r_{t-1}}{1-\gamma_1} \qquad v_t = \frac{(1-\gamma_2)\{G_i^t\}^2 + \gamma_2\, v_{t-1}}{1-\gamma_2}$$

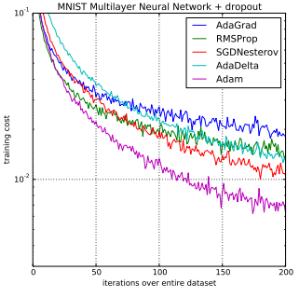
Parameter update given by:

$$w_i^t = w_i^{t-1} - \eta \frac{r_t}{\sqrt{v_t} + \epsilon}$$

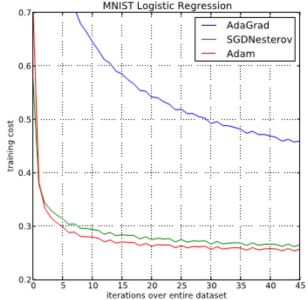
- RMSProp uses a momentum on the re-scaled gradient. Adam uses first and second moment of the gradient, also adds a bias-correction term.
- Adam can be understood as the combination of RMSprop (second moment) and SGD with momentum (first moment).
- Adam is probably one of the most popular optimizer in DL. In practice produces good results.



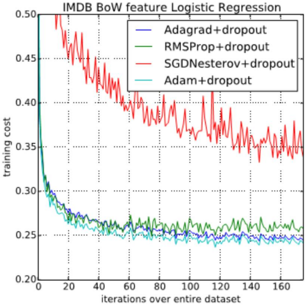
Results



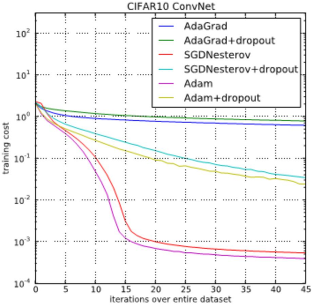
MNIST Multilayer Neural Network + dropout
AdaGrad
RMSProp
SGDNesterov
AdaDelta
Adam
training cost
10^{-1}
10^{-2}
0
50
100
150
200
iterations over entire dataset



MNIST Logistic Regression
AdaGrad
SGDNesterov
Adam
training cost
0.7
0.6
0.5
0.4
0.3
0.2
0
5
10
15
20
25
30
35
40
45
iterations over entire dataset



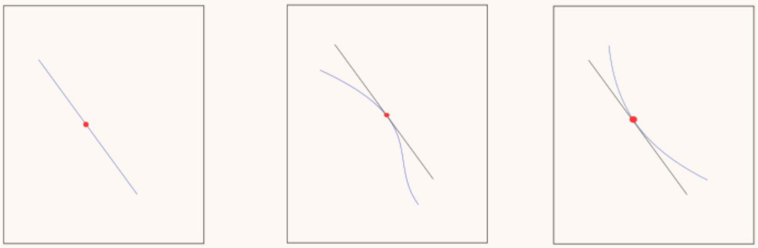
IMDB BoW feature Logistic Regression
Adagrad+dropout
RMSProp+dropout
SGDNesterov+dropout
Adam+dropout
training cost
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0
20
40
60
80
100
120
140
160
iterations over entire dataset



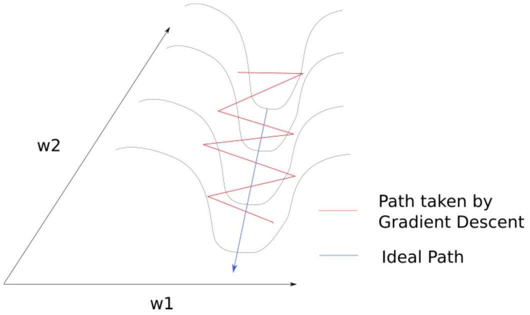
CIFAR10 ConvNet
AdaGrad
AdaGrad+dropout
SGDNesterov
SGDNesterov+dropout
Adam
Adam+dropout
training cost
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
0
5
10
15
20
25
30
35
40
45
iterations over entire dataset

What about using second order derivatives?

Problem with gradient descent

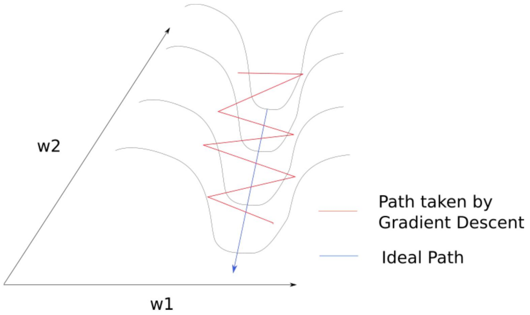


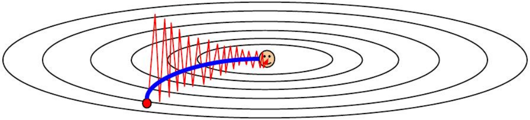
Problem with gradient descent



Gradient descent wastes time repeatedly descending canyon walls, because they are the steepest feature.

Problem with gradient descent

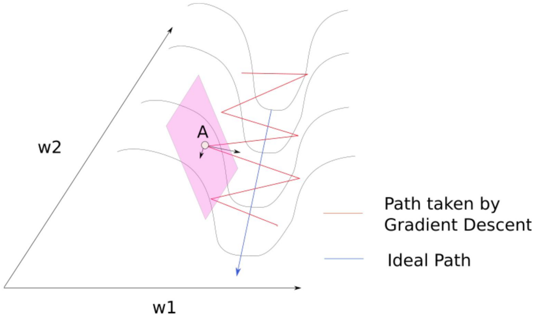




Slow progress along flat direction (valley), jitter along steep one.

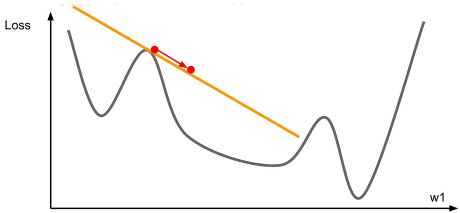
Problem with gradient descent

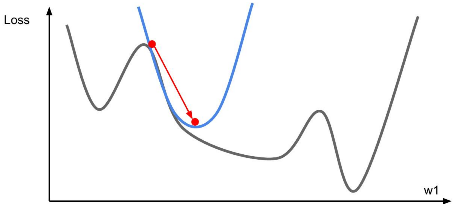
Gradient sees a local and planar approximation, it does not consider curvature.



- Directional derivative is rapidly changing
- An optimization algorithm based on 2nd derivatives could predict that the steepest direction is not really a promising search direction (why?).

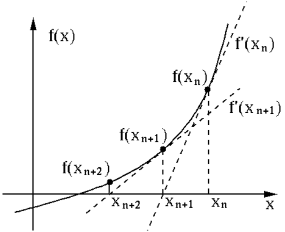
First vs second order derivatives





Second order gradient descent based methods

The famous Newton-Raphson method:



$$f(x_{n+1}) = f(x_n + \Delta_x) = f(x_n) + f'(x_n)\Delta_x + \ldots$$
$$f(x_{n+1}) = f(x_n + \Delta_x) \approx f(x_n) + f'(x_n)\Delta_x$$

Searching for a root at $x_0 + \Delta_x$:

$$x_{n+1} = x_n - \gamma \frac{f(x_n)}{f'(x_n)}$$

In the case of optimization, we are looking for the roots of $f'(x)$. Then we have:

$$x_{n+1} = x_n - \gamma \frac{f'(x_n)}{f''(x_n)}$$

Moving to high dimensions, we have to deal with the Jacobian and the Hessian matrixes:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \frac{\bigtriangledown f(x_n)}{Hf(x_n)}$$

$$\mathbf{x}_{n+1} = \ \mathbf{x}_n - Hf(x_n)^{-1} \bigtriangledown f(x_n)$$

Any problem with $Hf(x_n)^{-1}$? Matrix inversion is not trivial.

Conclusions

- SGD is a slow but secure (stable) road to the optimum (local/global).
- SGD has a global learning rate.
- Adaptive methods have a per-dimension learning rate. They are faster but less stable.
- Among adaptive techniques Adam is one of the most populars.
- Second-order methods make more clever progress toward the goal, but are more expensive and unstable.
- To avoid jitter close to the goal, use learning rate schedules. Automatically anneal the learning rate based on number of epochs (actually, in DL use minibatch steps).