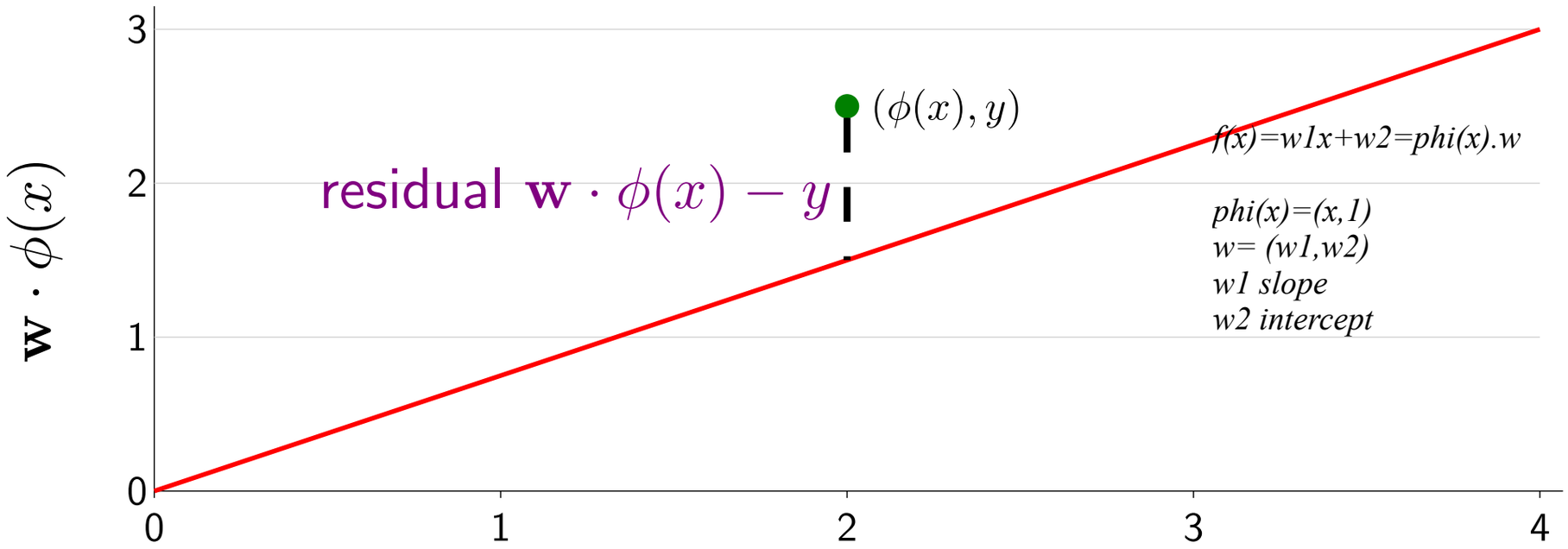
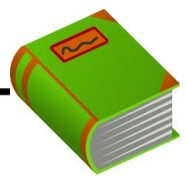
Large scale linear regression and gradient descendant optimization

Based on material of Stanford CS221 course Artificial Intelligence: Principles and Techniques

Linear regression

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$



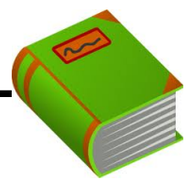


Definition: residual

The **residual** is $(\mathbf{w} \cdot \phi(x)) - y$, the amount by which prediction $f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$ overshoots the target y.

Linear regression

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$



Definition: squared loss

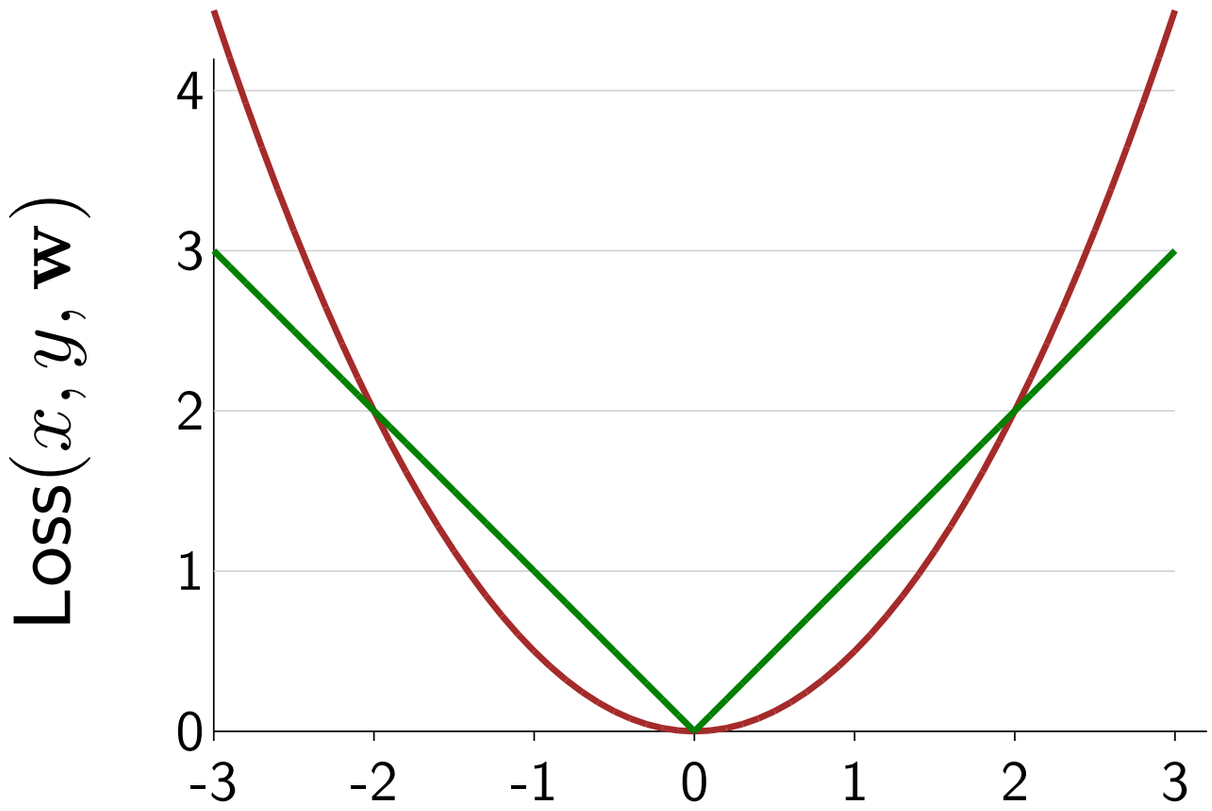
$$\text{Loss}_{\text{squared}}(x, y, \mathbf{w}) = (\underbrace{f_{\mathbf{w}}(x) - y}_{\text{residual}})^2$$

Example:

$$\mathbf{w} = [2, -1], \phi(x) = [2, 0], y = -1$$

$$\text{Loss}_{\text{squared}}(x, y, \mathbf{w}) = 25$$

Regression loss functions



residual $(\mathbf{w} \cdot \phi(x)) - y$

$\text{Loss}_{\text{squared}}(x, y, \mathbf{w}) = (\mathbf{w} \cdot \phi(x) - y)^2$

$\text{Loss}_{\text{absdev}}(x, y, \mathbf{w}) = |\mathbf{w} \cdot \phi(x) - y|$

Loss minimization framework

So far: one example, $\text{Loss}(x, y, \mathbf{w})$ is easy to minimize.

Key idea: minimize training loss

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

$$\min_{\mathbf{w} \in \mathbb{R}^d} \text{TrainLoss}(\mathbf{w})$$

Key: need to set $\mathbf{w}$ to make global tradeoffs — not every example can be happy.

Which regression loss to use?

Example: $\mathcal{D}_{\text{train}} = \{(1, 0), (1, 2), (1, 1000)\}$, $\phi(x) = x$

For least squares (L_2) regression:

$$\text{Loss}_{\text{squared}}(x, y, \mathbf{w}) = (\mathbf{w} \cdot \phi(x) - y)^2$$

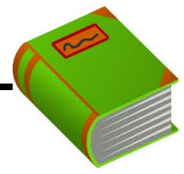
- $\mathbf{w}$ that minimizes training loss is **mean** y
- Mean: tries to accommodate every example, popular

For least absolute deviation (L_1) regression:

$$\text{Loss}_{\text{absdev}}(x, y, \mathbf{w}) = |\mathbf{w} \cdot \phi(x) - y|$$

- $\mathbf{w}$ that minimizes training loss is **median** y
- Median: more robust to outliers

How to optimize?



Definition: gradient

The gradient $\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$ is the direction that increases the loss the most.

Algorithm: gradient descent

Initialize $\mathbf{w} = [0, \ldots, 0]$

For $t = 1, \ldots, T$:

$$\mathbf{w} \leftarrow \mathbf{w} - \underbrace{\eta}_{\text{step size}} \underbrace{\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})}_{\text{gradient}}$$

Least squares regression

Objective function:

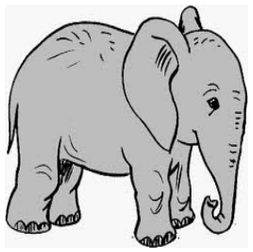
$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} (\mathbf{w} \cdot \phi(x) - y)^2$$

Gradient (use chain rule):

$$\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} 2(\underbrace{\mathbf{w} \cdot \phi(x) - y}_{\text{prediction} - \text{target}})\phi(x)$$

reduce (action)

Map



Gradient descent is slow

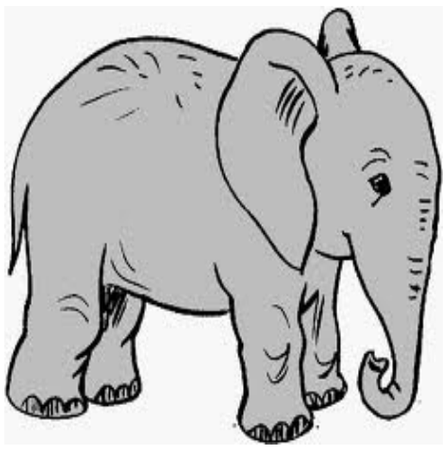
$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

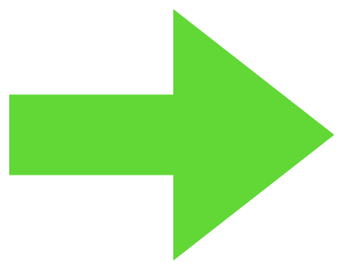
Gradient descent:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$$

Problem: each iteration requires going over all training examples — expensive when have lots of data!

First solution





Lab session part 1 :

Implement LR with DG in Spark, and test it on diamonds data.

Stochastic gradient descent

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

Gradient descent (GD):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$$

Stochastic gradient descent (SGD):

For each $(x, y) \in \mathcal{D}_{\text{train}}$:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$$

Key idea: stochastic updates

It's not about **quality**, it's about **quantity**.

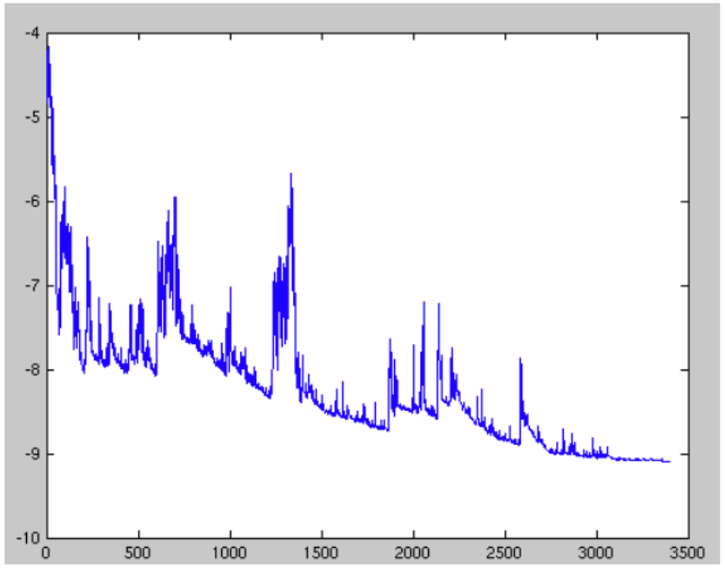
Lab session part 2:

Implement SGD in Spark ?
How the sequential aspect can be dealt with in Spark ?

Minibatch GD

Main assumptions

- Batch GD has de disadvantage of being slow
- Also many points in the training set are very similar so we risk to have high redundancy in the gradient computation at each step
- SGD provides a first solution, but it is characterised by high fluctuations
- Mini-Batch GD provides a first solution
  - Largely used in DeepLearning training
  - It can leverage efficient algorithms for matrix multiplication, performed for each batch



https://upload.wikimedia.org/wikipedia/commons/f/f3/Stogra.png

Mini-Batch GD

- Gradient parameters are updated for every mini-batch of n training exemples

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

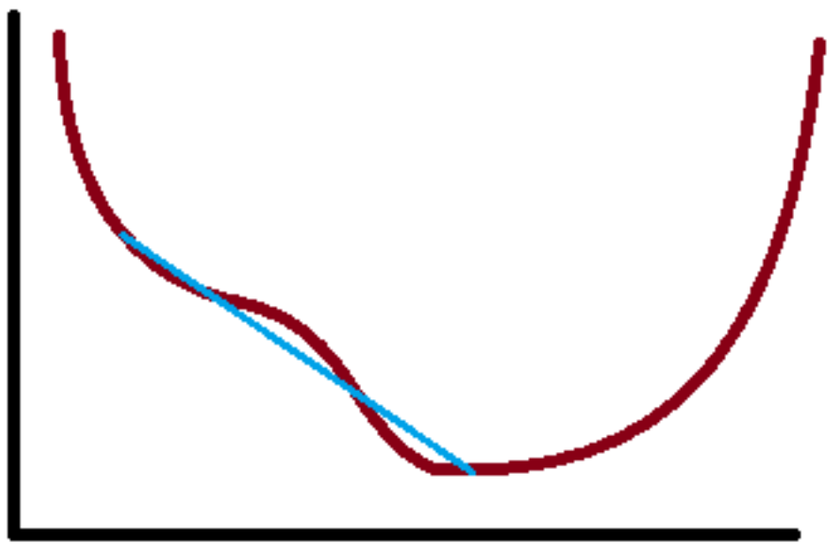
```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Challenges

- Finding a proper learning rate (lr)
  - small lr is good for convergence
  - high lr faster convergence, that can be hindered by much more fluctuations
- Additionally, the same learning rate is applied to all gradient parameters
  - in some cases this is not effective
  - some feature should contribute less/more to the learning process
  - for instance we would like that rarely occurring features would actually contribute to the learning, by means of larger local lr / parameter update
- We want minimise highly non-convex functions

Quasi convex functions

- Local minima correspond to global minima
- But GD can get stuck on stationary points
    - fluctuations ensured by SGD and mini-batch helps in not getting stuck on these point
    - another approach is the heavy-ball method, that we are going to see

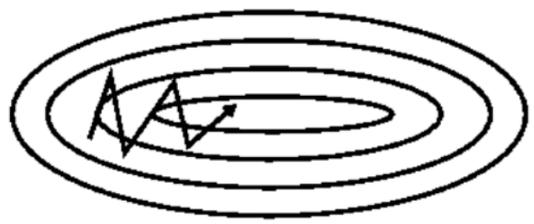


More efficient algorithms

Momentum
heavy ball method

- It helps in speeding up SGD and to attenuate oscillations

(a) SGD without momentum



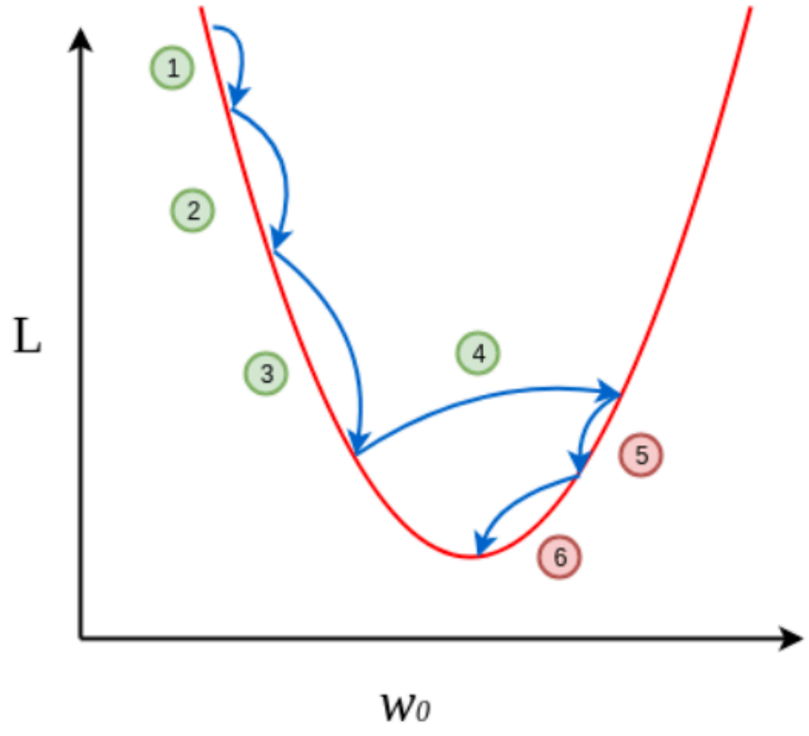
(b) SGD with momentum

- This is ensured by adding a fraction of the update vector of the previous iteration to the current update vector

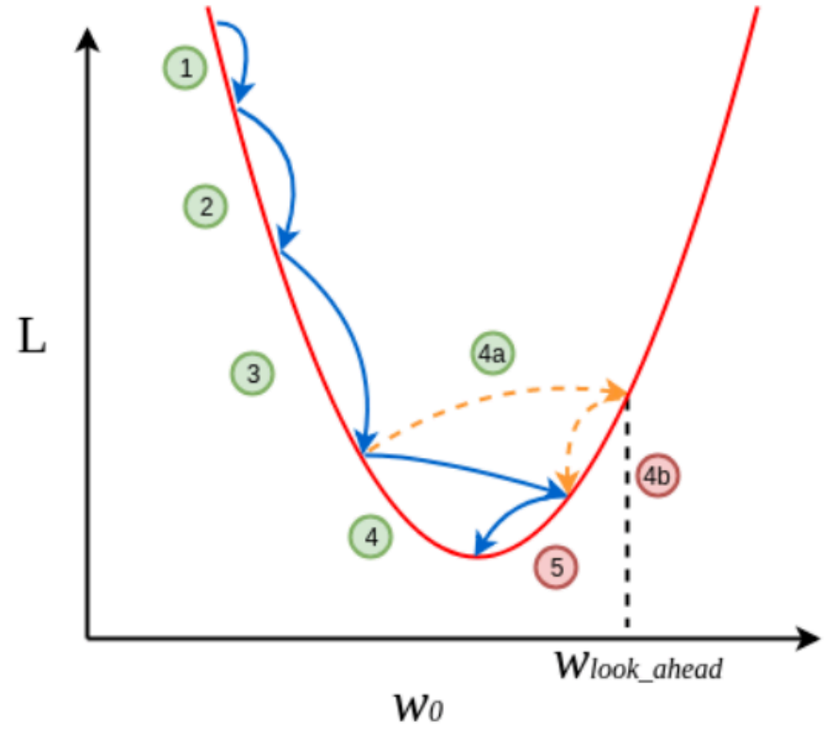
$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

- *Momentum term*
  usually set close to 0.9

Comparison



(a) Momentum-Based Gradient Descent



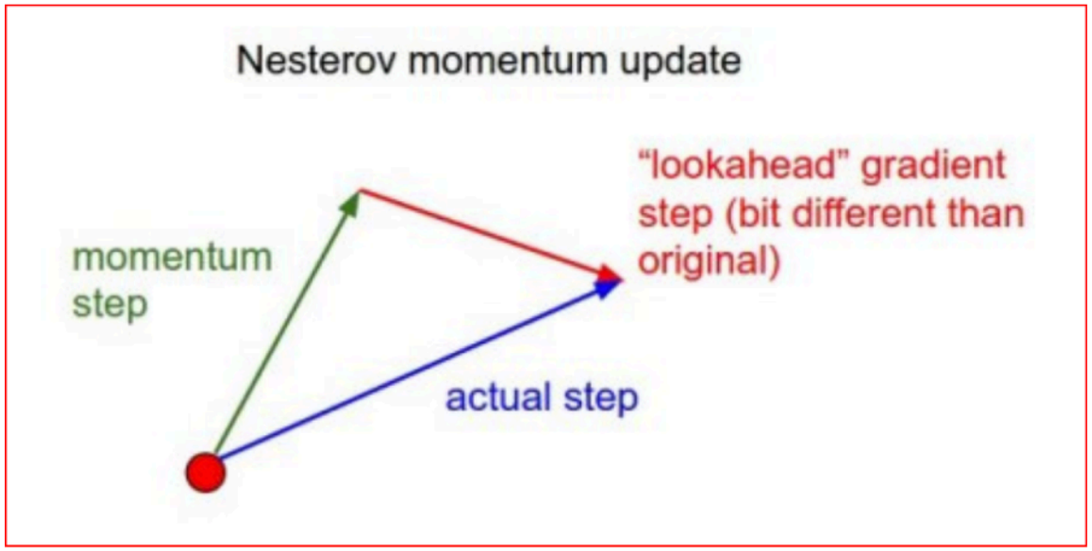
(b) Nesterov Accelerated Gradient Descent

Nesterov accelerated gradient

- Making tha rolling ball smarter by some kind of look-ahead

- In case the gradient suddenly increases, this approach allows to slow down in order to increase likelihood not to be trapped in local minima

- Generally better than momentum



source:http://cs231n.github.io/neural-networks-3/

$$\theta = \theta - v_t$$
$$v_t = \gamma v_{t-1} + \eta \nabla J(\theta - \gamma v_{t-1})$$

$\theta - \gamma v_{t-1}$ is the gradient of looked ahead

Nesterov Accelerated Gradient

Adagrad

- It adapts the learning rate to the parameters
- **Performing larger updates for infrequent parameters, and smaller updates for frequent ones.**
- Dealing with millions of parameters, as in Deep Learning
- Autotuning of parameters
- Works well with the high-dimensional non-convex nature of neural networks optimization, where the learning rate could be too small in some dimension and too large in another dimension.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\varepsilon I + diag(G_t)}} \cdot g_t,$$

$$g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(x^{(i)}, y^{(i)}, \theta_t),$$

$$G_t = \sum_{\tau=1}^{t} g_\tau g_\tau^\top$$

More in detail

$$\begin{bmatrix} \theta_{t+1}^{(1)} \\ \theta_{t+1}^{(2)} \\ \vdots \\ \theta_{t+1}^{(m)} \end{bmatrix} = \begin{bmatrix} \theta_{t}^{(1)} \\ \theta_{t}^{(2)} \\ \vdots \\ \theta_{t}^{(m)} \end{bmatrix} - \eta \left(\begin{bmatrix} \varepsilon & 0 & \cdots & 0 \\ 0 & \varepsilon & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \varepsilon \end{bmatrix} + \begin{bmatrix} G_t^{(1,1)} & 0 & \cdots & 0 \\ 0 & G_t^{(2,2)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & G_t^{(m,m)} \end{bmatrix} \right)^{-1/2} \cdot \begin{bmatrix} g_t^{(1)} \\ g_t^{(2)} \\ \vdots \\ g_t^{(m)}, \end{bmatrix} \tag{4}$$

$$G_t = \sum_{\tau=1}^{t} g_\tau g_\tau^\top \qquad g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(x^{(i)}, y^{(i)}, \theta_t),$$

More in detail

$$\begin{bmatrix} \theta_{t+1}^{(1)} \\ \theta_{t+1}^{(2)} \\ \vdots \\ \theta_{t+1}^{(m)} \end{bmatrix} = \begin{bmatrix} \theta_{t}^{(1)} \\ \theta_{t}^{(2)} \\ \vdots \\ \theta_{t}^{(m)} \end{bmatrix} - \begin{bmatrix} \frac{\eta}{\sqrt{\varepsilon + G_t^{(1,1)}}} & 0 & \cdots & 0 \\ 0 & \frac{\eta}{\sqrt{\varepsilon + G_t^{(2,2)}}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\eta}{\sqrt{\varepsilon + G_t^{(m,m)}}} \end{bmatrix} \cdot \begin{bmatrix} g_t^{(1)} \\ g_t^{(2)} \\ \vdots \\ g_t^{(m)} \end{bmatrix} \quad (5)$$

$$G_t = \sum_{\tau=1}^{t} g_\tau g_\tau^\top \qquad g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(x^{(i)}, y^{(i)}, \theta_t),$$

It’s now coding time in Spark of all the methods seen so far