

DEEP LEARNING FOR COMPUTER VISION

Summer Seminar UPC TelecomBCN, 4 - 8 July 2016



Instructors



Xavier
Giró-i-Nieto



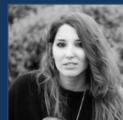
Elisa
Sayrol



Amaia
Salvador



Jordi
Torres



Eva
Mohedano



Kevin
McGuinness

Organizers



UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH



**Barcelona
Supercomputing
Center**
Centro Nacional de Supercomputación



Dublin City University
Oileici Chathair Bhaile Átha Cliath



Centre for Data Analytics



GPU
CENTER OF
EXCELLENCE

Co-funded by the
Erasmus+ Programme
of the European Union



+ info: TelecomBCN.DeepLearning.Barcelona

Day 3 Lecture 3

Optimizing deep networks

Convex optimization

A function is convex if for all $\alpha \in [0,1]$:

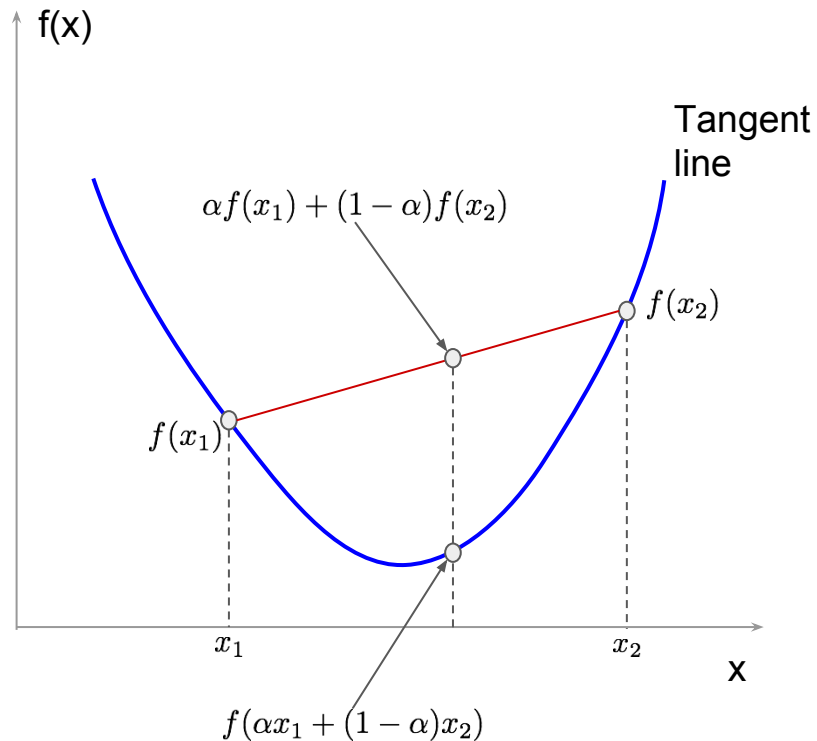
$$\alpha f(x_1) + (1 - \alpha)f(x_2) \geq f(\alpha x_1 + (1 - \alpha)x_2)$$

Examples

- Quadratics
- 2-norms

Properties

- Local minimum is global minimum



Gradient descent

E.g. linear regression

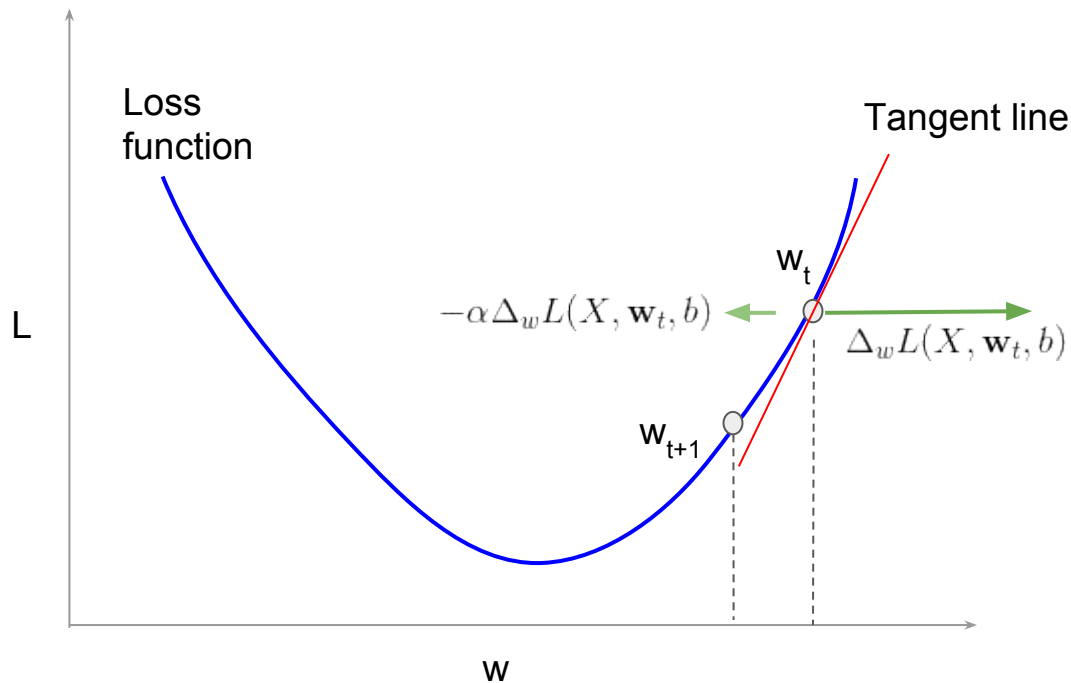
$$L(X, y, \mathbf{w}, b) = \frac{1}{2N} \sum_{i=1}^N (y_i - f(x_i, \mathbf{w}, b))^2$$

Need to **optimize L**

Gradient descent

$$\Delta_w L = \frac{1}{N} \sum_{i=1}^N (f(x_i) - y_i) x_i$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \Delta_w L(X, \mathbf{w}_t, b)$$



Stochastic gradient descent

- Computing the gradient for the full dataset at each step is slow
 - Especially if the dataset is large
- Note:
 - For many loss functions we care about, the gradient is the average over losses on individual examples

$$L(X, y, \mathbf{w}, b) = \frac{1}{2N} \sum_{i=1}^N (y_i - f(x_i, \mathbf{w}, b))^2$$

- Idea:
 - Pick a **single random training example**
 - **Estimate** a (noisy) loss on this single training example (the *stochastic* gradient)
 - Compute gradient wrt. this loss
 - Take a step of gradient descent using the estimated loss

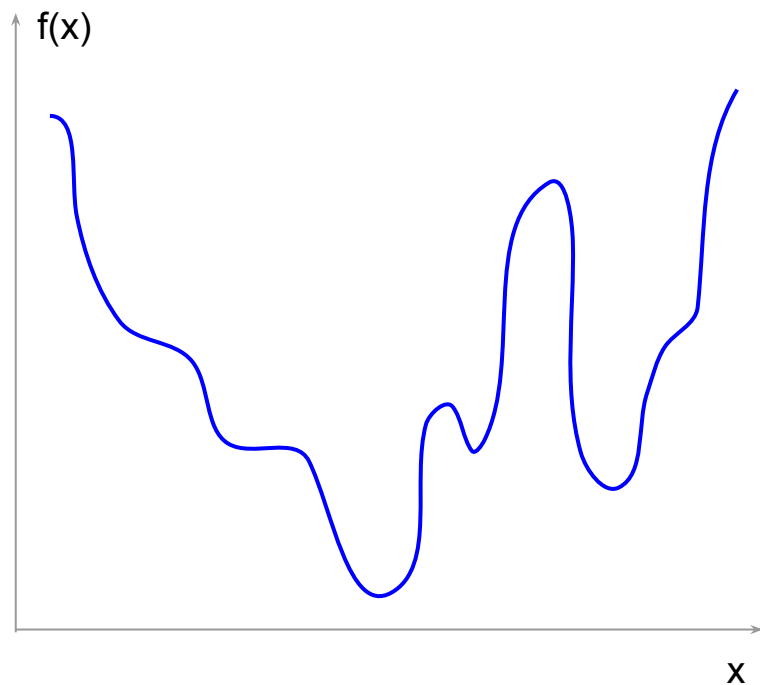
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \Delta_{\mathbf{w}} L(X, \mathbf{w}_t, b)$$

Non-convex optimization

Objective function in deep networks is non-convex

- May be many local minima
- Plateaus: flat regions
- Saddle points

Q: Why does SGD seem to work so well for optimizing these complex non-convex functions??



Local minima

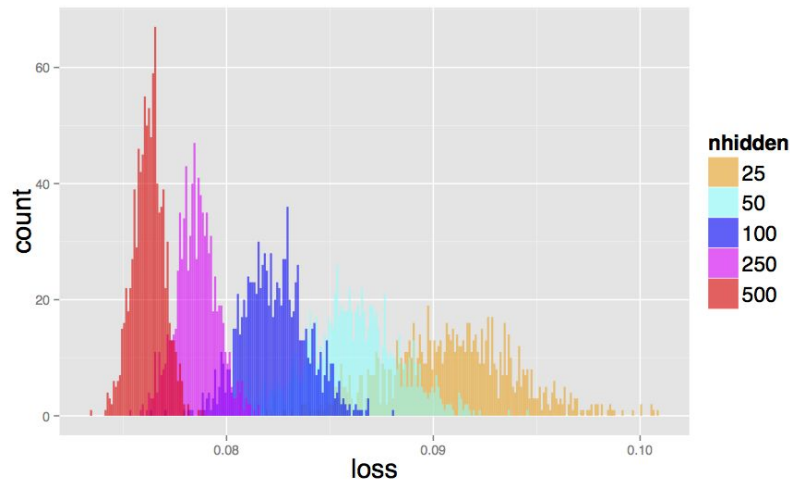
Q: Why doesn't SGD get stuck at local minima?

A: It does.

But:

- Theory and experiments suggest that for high dimensional deep models, value of loss function at most local minima is close to value of loss function at global minimum.

Most local minima are good local minima!



Value of local minima found by running SGD for 200 iterations on a simplified version of MNIST from different initial starting points. As number of parameters increases, local minima tend to cluster more tightly.

Saddle points

Q: Are there many saddle points in high-dimensional loss functions?

A: Local minima dominate in low dimensions, but **saddle points dominate in high dimensions.**

Why?

Eigenvalues of the Hessian matrix

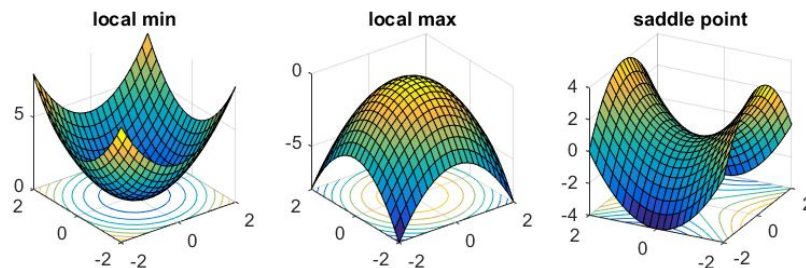
$$\mathbf{H}_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j}.$$

Intuition

Random matrix theory: $P(\text{eigenvalue} > 0) \sim 0.5$

At a critical point (zero grad) N dimensions means we need N positive eigenvalues to be local min.

As N grows it becomes exponentially unlikely to randomly pick all eigenvalues to be positive or negative, and therefore most critical points are saddle points.



Saddle points

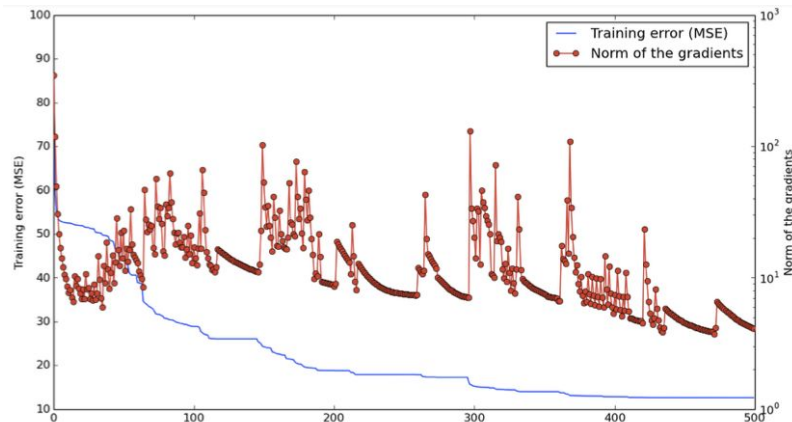
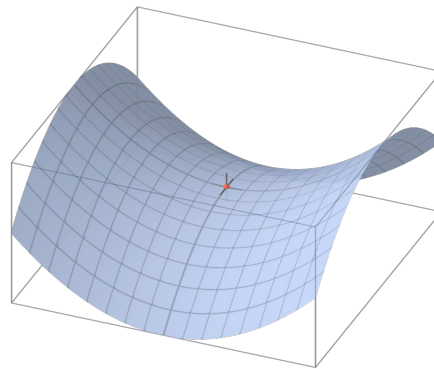
Q: Does SGD get stuck at saddle points?

A: No, not really

Gradient descent is initially attracted to saddle points, but unless it hits the critical point exactly, it will be repelled when close.

Hitting critical point exactly is unlikely: estimated gradient of loss is stochastic

Warning: Newton's method works poorly for neural nets as it is attracted to saddle points



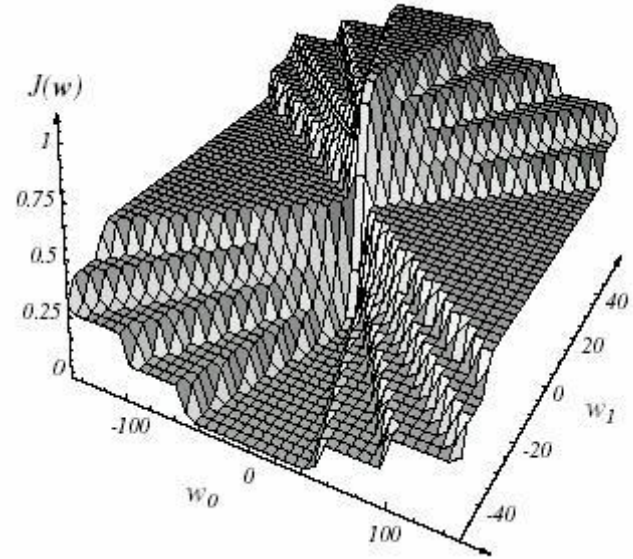
SGD tends to oscillate between slowly approaching a saddle point and quickly escaping from it

Plateaus

Regions of the weight space where loss function is mostly flat (small gradients).

Can sometimes be avoided using:

- Careful initialization
- Non-saturating transfer functions
- Dynamic gradient scaling
- Network design
- Loss function design



Learning rates and initialization

Choosing the learning rate

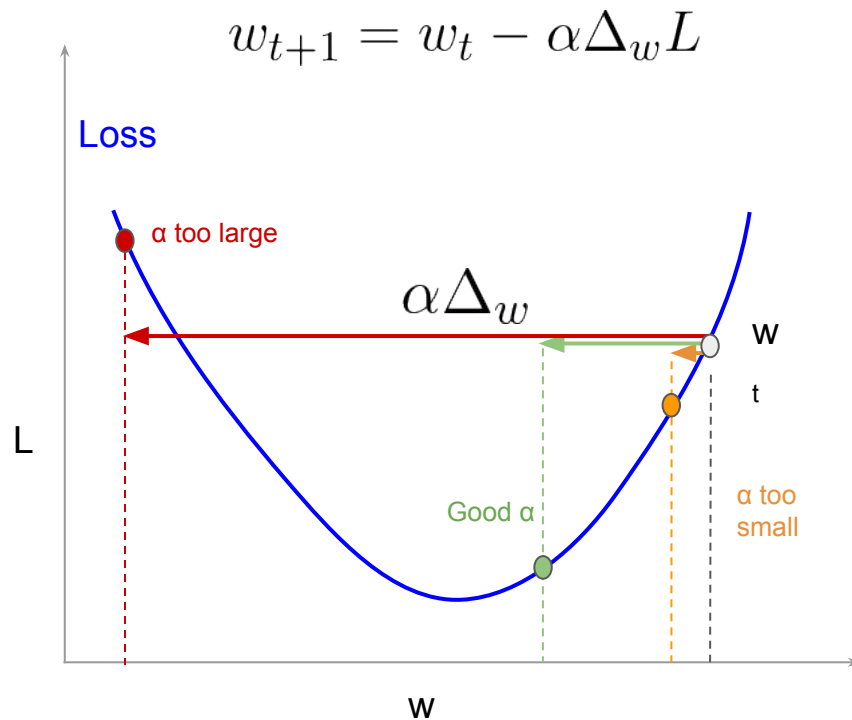
For most **first order** optimization methods, we need to choose a learning rate (aka **step size**)

- Too large: overshoots local minimum, loss increases
- Too small: makes very slow progress, can get stuck
- Good learning rate: makes steady progress toward local minimum

Usually want a higher learning rate at the start and a lower one later on.

Common strategy in practice:

- Start off with a high LR (like 0.1 - 0.001),
- Run for several **epochs** (1 - 10)
- Decrease LR by multiplying a constant factor (0.1 - 0.5)



Weight initialization

Need to pick a starting point for gradient descent: an initial set of weights

Zero is a very **bad idea!**

- Zero is a **critical point**
- Error signal will not propagate
- Gradients will be zero: no progress

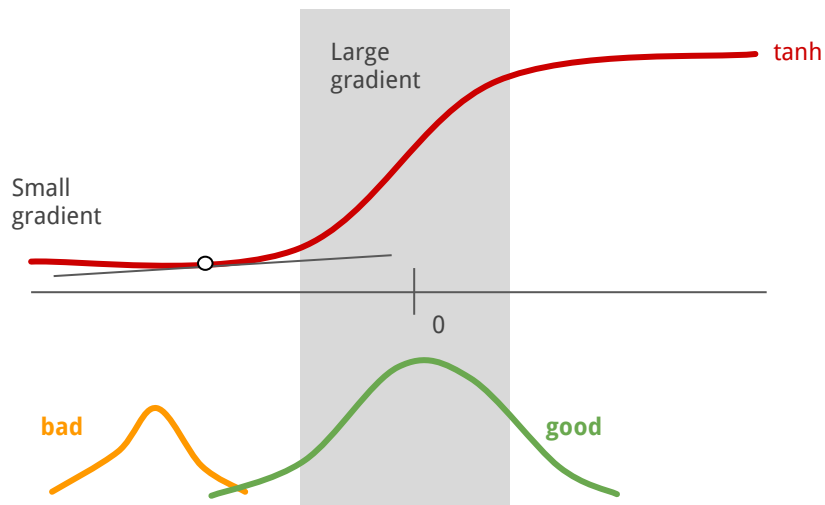
Constant value also bad idea:

- Need to break symmetry

Use **small random values:**

- E.g. zero mean Gaussian noise with constant variance

Ideally we want inputs to activation functions (e. g. sigmoid, tanh, ReLU) to be mostly **in the linear area** to allow larger gradients to propagate and converge faster.



Batch normalization

As learning progresses, the distribution of layer inputs changes due to parameter updates.

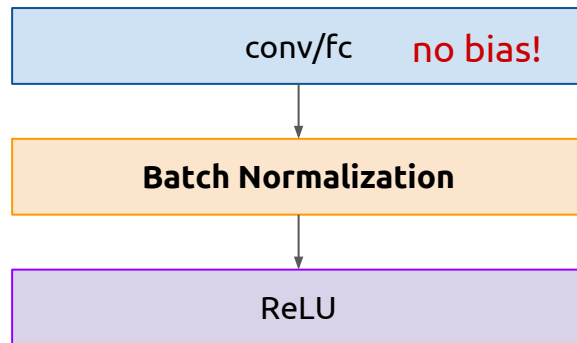
This can result in most inputs being in the nonlinear regime of the activation function and slow down learning.

Batch normalization is a technique to reduce this effect.

Works by re-normalizing layer inputs to have zero mean and unit standard deviation with respect to running batch estimates.

Also adds a **learnable scale and bias** term to allow the network to still use the nonlinearity.

Usually **allows much higher learning rates!**



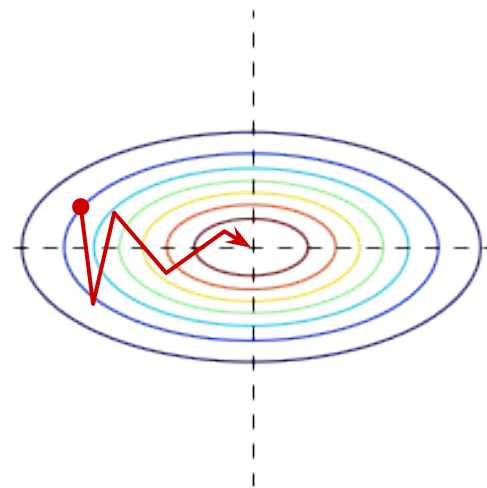
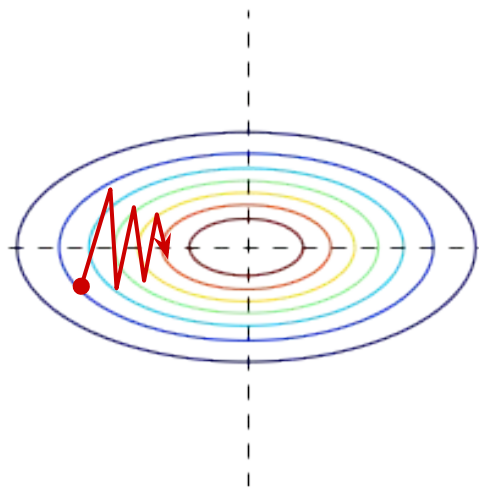
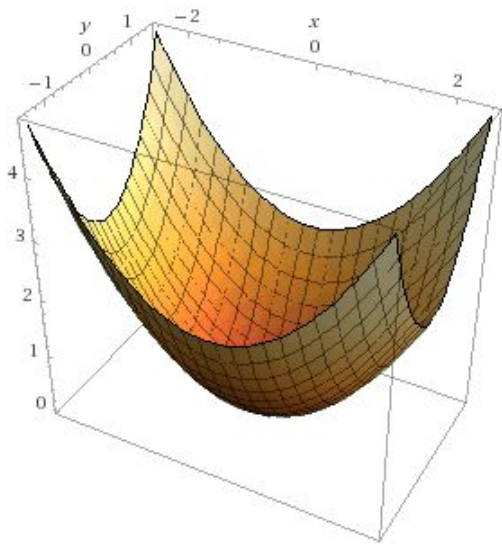
First-order optimization algorithms

(SGD bells and whistles)

Vanilla mini-batch SGD

$$\theta_t = \theta_{t-1} - \alpha \underbrace{\Delta_{\theta} L(\theta_{t-1})}_{\text{Evaluated on a mini-batch}}$$

Momentum



$$v_t = \gamma v_{t-1} + \alpha \Delta_{\theta} L(\theta_{t-1})$$

$$\theta_t = \theta_{t-1} - v_t$$

2x memory for parameters!


Nesterov accelerated gradient (NAG)

Approximate what the parameters will be on the next time step by using the current velocity.

Update the velocity using gradient where we predict we will be, instead of where we are now.

$$v_t = \gamma v_{t-1} + \alpha \Delta_{\theta} L(\theta_{t-1} - \gamma v_{t-1})$$

$$\theta_t = \theta_{t-1} - v_t$$



What we expect the parameters to be based on momentum alone

Adagrad


Adapts the learning rate for each of the parameters based on sizes of previous updates.

- Scales updates to be larger for parameters that are updated less
- Scales updates to be smaller for parameters that are updated more

Store sum of squares of gradients so far in diagonal of matrix G_t

$$G_t = \sum_{i=0}^t \text{diag}(\Delta_{\theta} L(\theta)_i)^2$$

Gradient of loss at timestep i



Update rule: $\theta_t = \theta_{t-1} - \alpha G_t^{-\frac{1}{2}} \Delta_{\theta} L(\theta)$

RMSProp

Modification of Adagrad to address aggressively decaying learning rate.

Instead of storing sum of squares of gradient over all time steps so far, use a **decayed moving average** of sum of squares of gradients

$$G_t = \gamma G_t + (1 - \gamma) \text{diag}(\Delta_{\theta} L(\theta))^2$$

Update rule: $\theta_t = \theta_{t-1} - \alpha G_t^{-\frac{1}{2}} \Delta_{\theta} L(\theta)$

Adam

Combines momentum and RMSProp

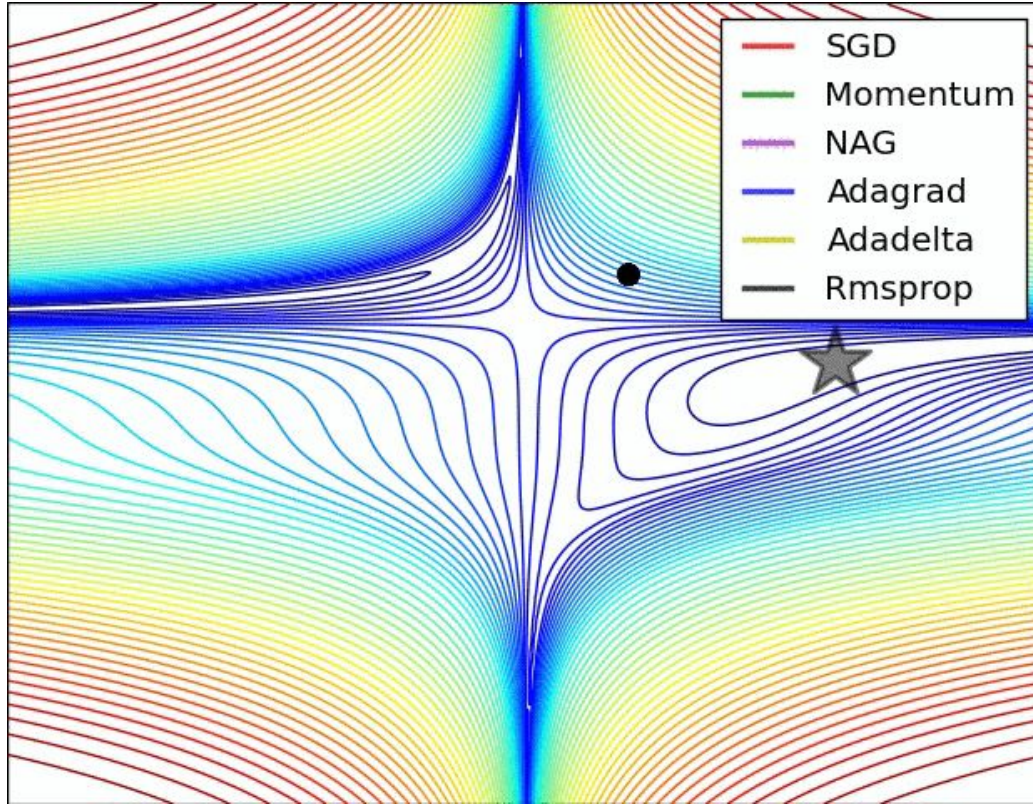
Keep decaying average of both first-order moment of gradient (momentum) and second-order moment (like RMSProp)

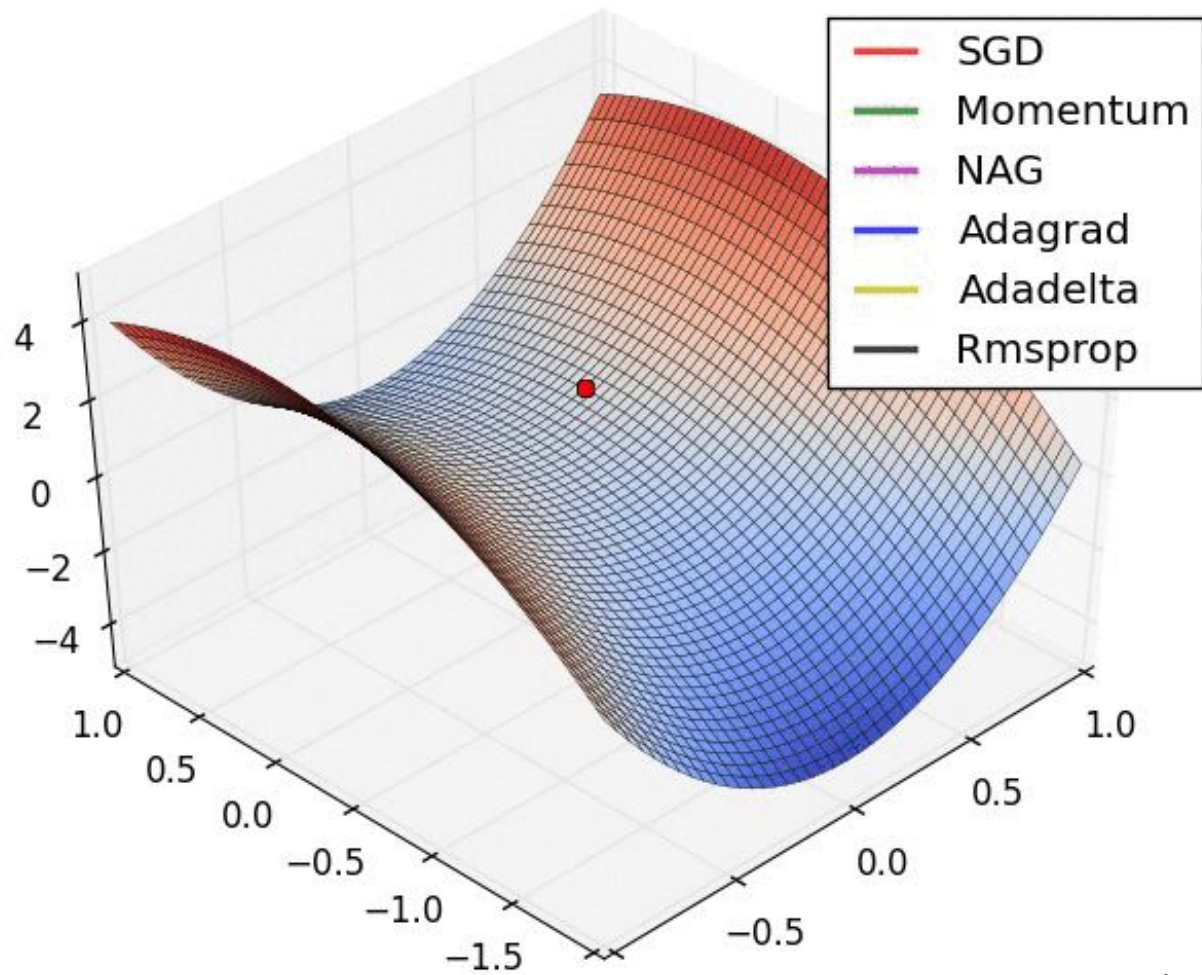
First-order: $v_t = \gamma_1 v_{t-1} + (1 - \gamma_1) \Delta_{\theta} L(\theta)$

Second-order: $G_t = \gamma_2 G_{t-1} + (1 - \gamma_2) \text{diag}(\Delta_{\theta} L(\theta))^2$

Update rule: $\theta_t = \theta_{t-1} - \alpha G^{-\frac{1}{2}} v_t$

3x memory!





Summary

Non-convex optimization means local minima and saddle points

In high dimensions, there are many more saddle points than local optima

Saddle points attract, but usually SGD can escape

Choosing a good learning rate is critical

Weight initialization is key to ensuring gradients propagate nicely (also batch normalization)

Several SGD extensions that can help improve convergence