Numerical Optimization

Ezepov Ilya

Agenda

- Optimization in ML
- Problems with optimization in ML
- Approaches to optimisation
- 50 shades of Gradient Descent

Optimization in ML

What is machine learning?

- Given the data **X**
  N row(data points), M columns (features)

- and the correct “answers” **y**

- Find the best function **f**

- Such that **f(X)** will be close to **y**

Finding the function

- How to select a good function?
- In general, we are doomed
- Let’s select from a class of function and tune the parameters

Function classes a.k.a machine learning algorithms

Algo	Parameters
Naive bayess	P(x|y) distribution
Linear regression	M Coefficients
K means	Clusters centeres
SVM	M Coefficients + Kernel parameters
Decision Trees	Split coordinates, predictions in leafs
Neural Networks	A lot of coefficients

Here comes the optimization

- We need to choose **w** in **f(X, w)**
- to minimize loss function **loss(f(X, w), y)**
- Optimization: minimize **g** == maximize **-g**

Problems

1. Curse of dimensionality

- In high dimensions things work different
- 100 points uniformly placed on the unit 1-dimensional cube will cover with gaps no more than 10^{-2}
- Then we have 9 more dimensions. How many data points do we need to cover it with the same density?

$$10^{20}$$

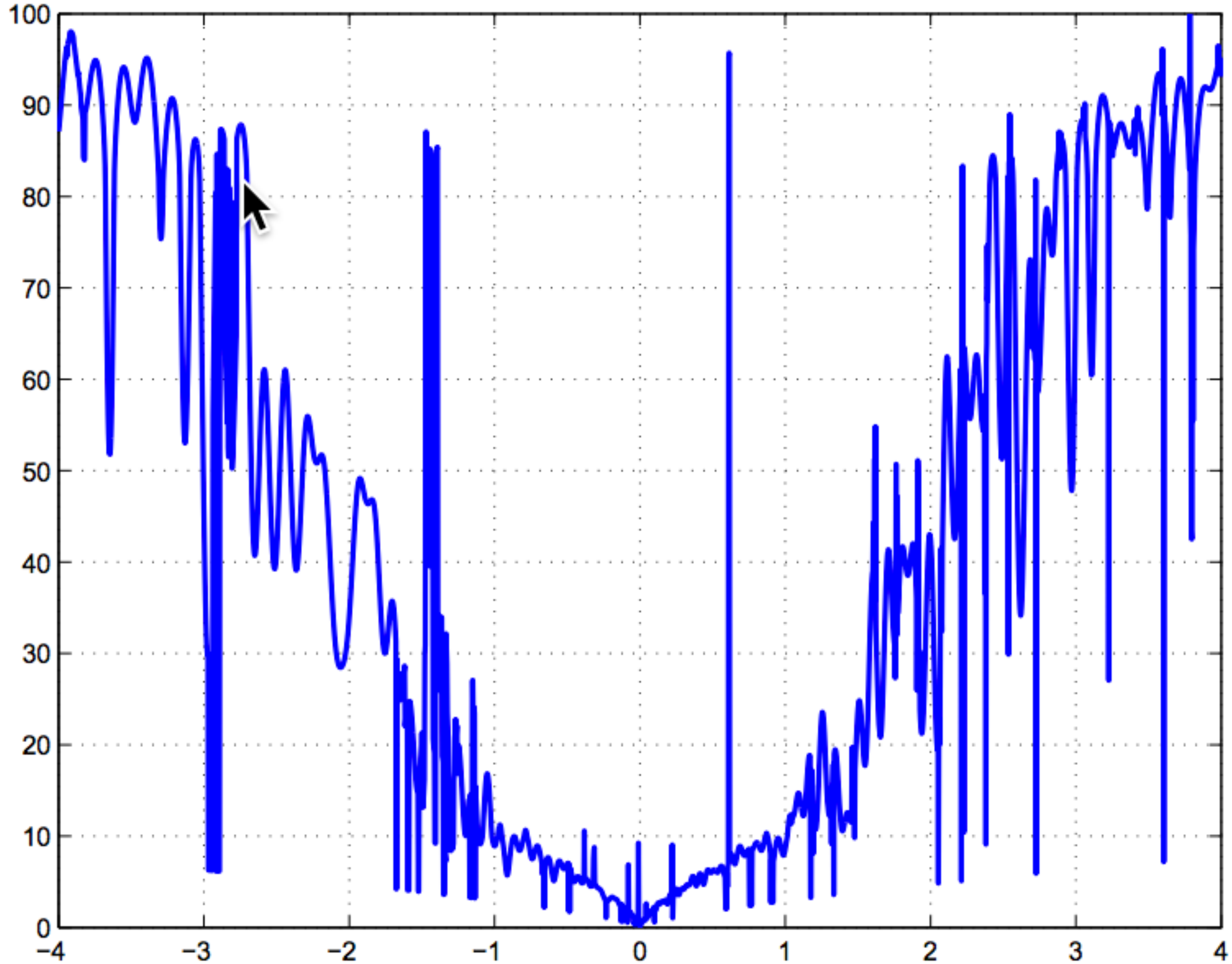
1. Curse of dimensionality

- In my current work I need to train neural network with **164 480** parameters
- Impossible to create a good search policy
- Brute search is absolutely impossible

2. Non separable problmes

- Separable:
  argmin $f(w_1, w_2)$ = argmin $f(w_1, \ldots)$, argmin $f(\ldots, w_2)$
- ML optimization is non-separable, parameters are dependent

3. Function itself



Approaches to optimization on linear model

Linear models

- Data **X** with constant columns, real value answer **y**
- **f(X, w) = XW**
- MSE loss

Explicit Solution (OLS)

$$XW = \hat{y}$$

$$\frac{1}{2}(y - XW)(y - XW)^T \rightarrow min$$

$$W = (X^T X)^{-1} X^T y$$

Explicit Solution

$$XW = \hat{y}$$

$$\frac{1}{2}(y - XW)(y - XW)^T \rightarrow min$$

$$W = (X^T X)^{-1} X^T y$$

Matrix multiplication/inversion:

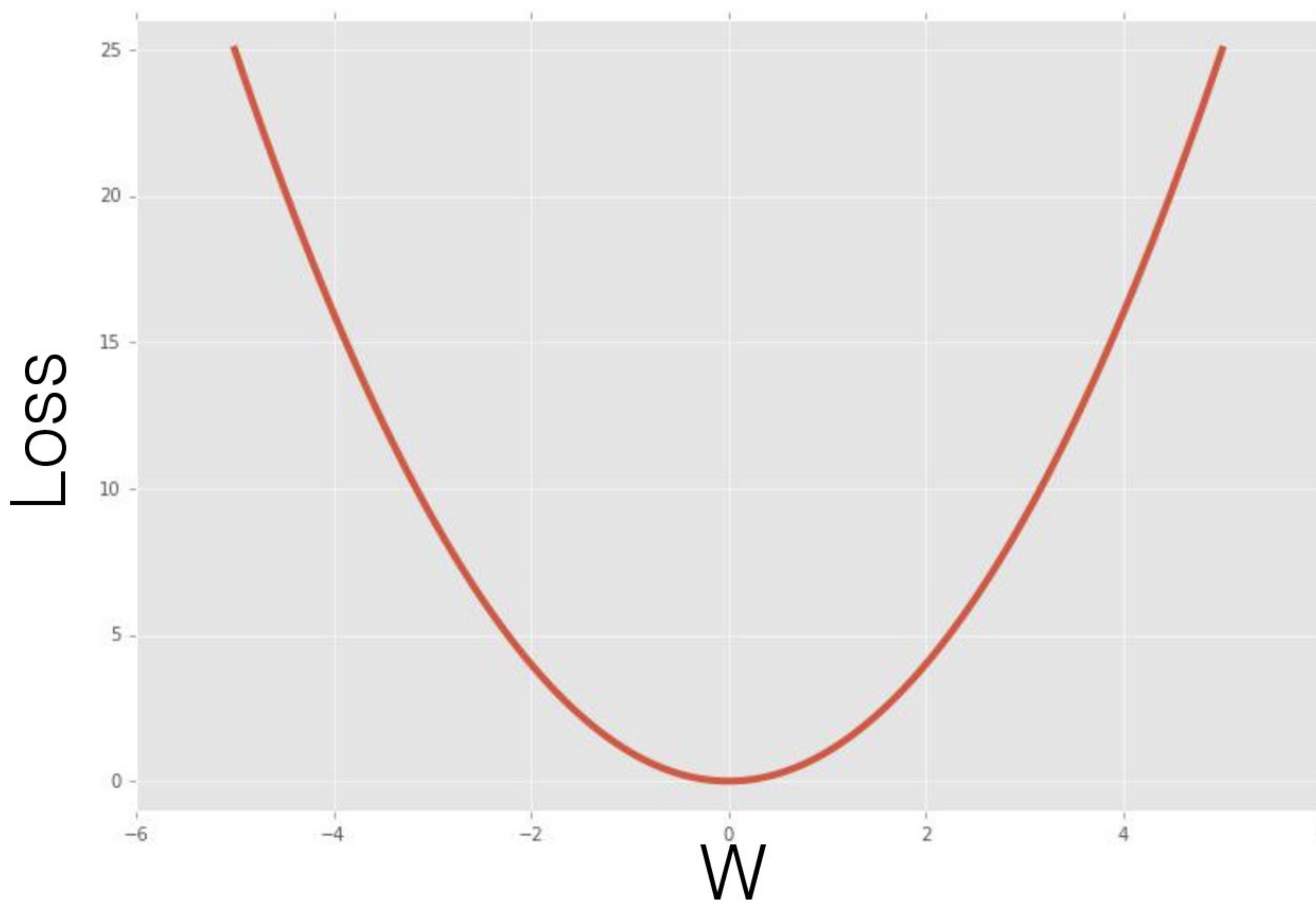
Coppersmith–Winograd algorithm, $O(n^{2.373})$

Gradient Descent

Gradient Descent

$$Loss(W) = \frac{1}{2}(y - XW)(y - XW)^T$$

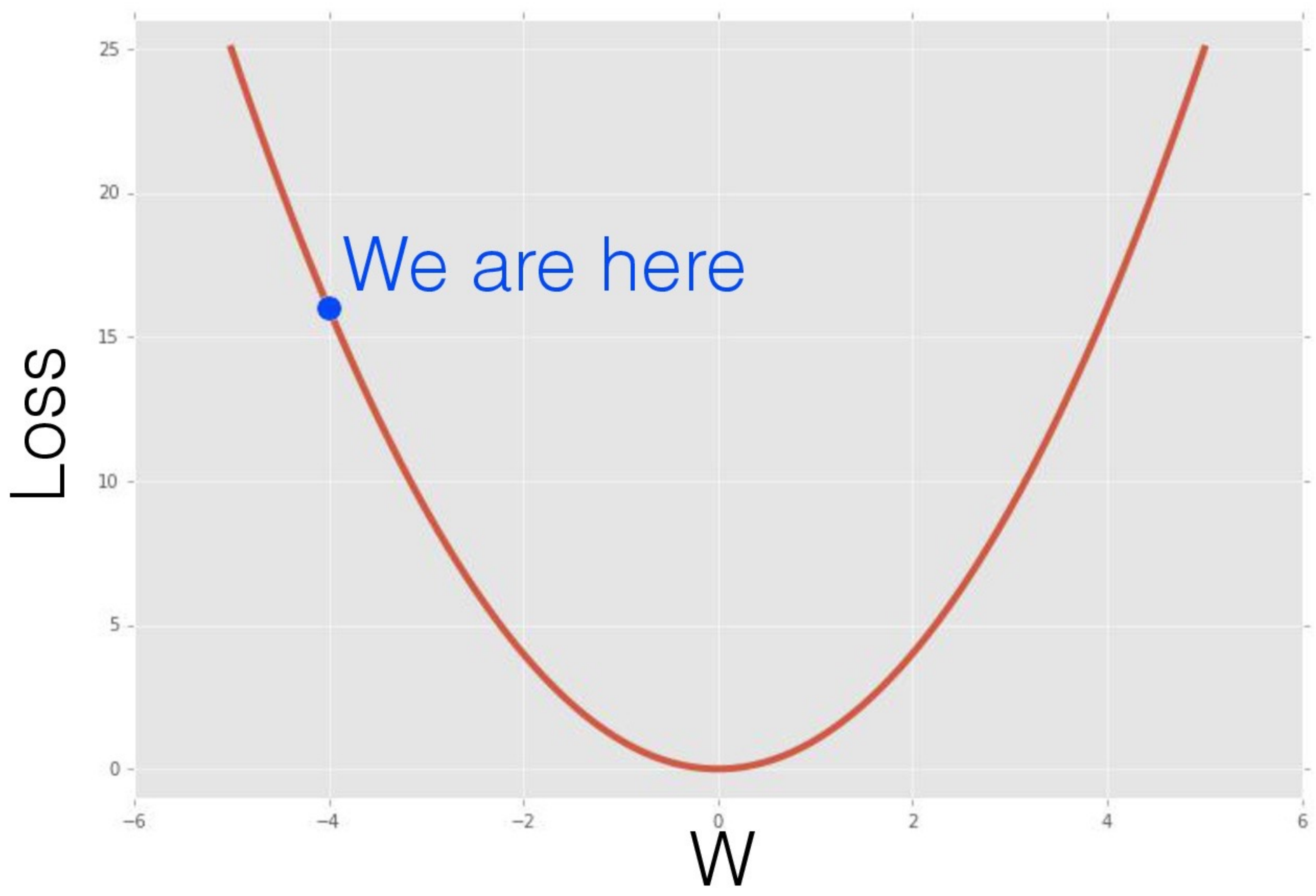
$$\frac{\partial Loss(W)}{\partial W} = ?$$



Gradient Descent

$$W_t = W_{t-1} - \alpha \nabla W_{t-1}$$

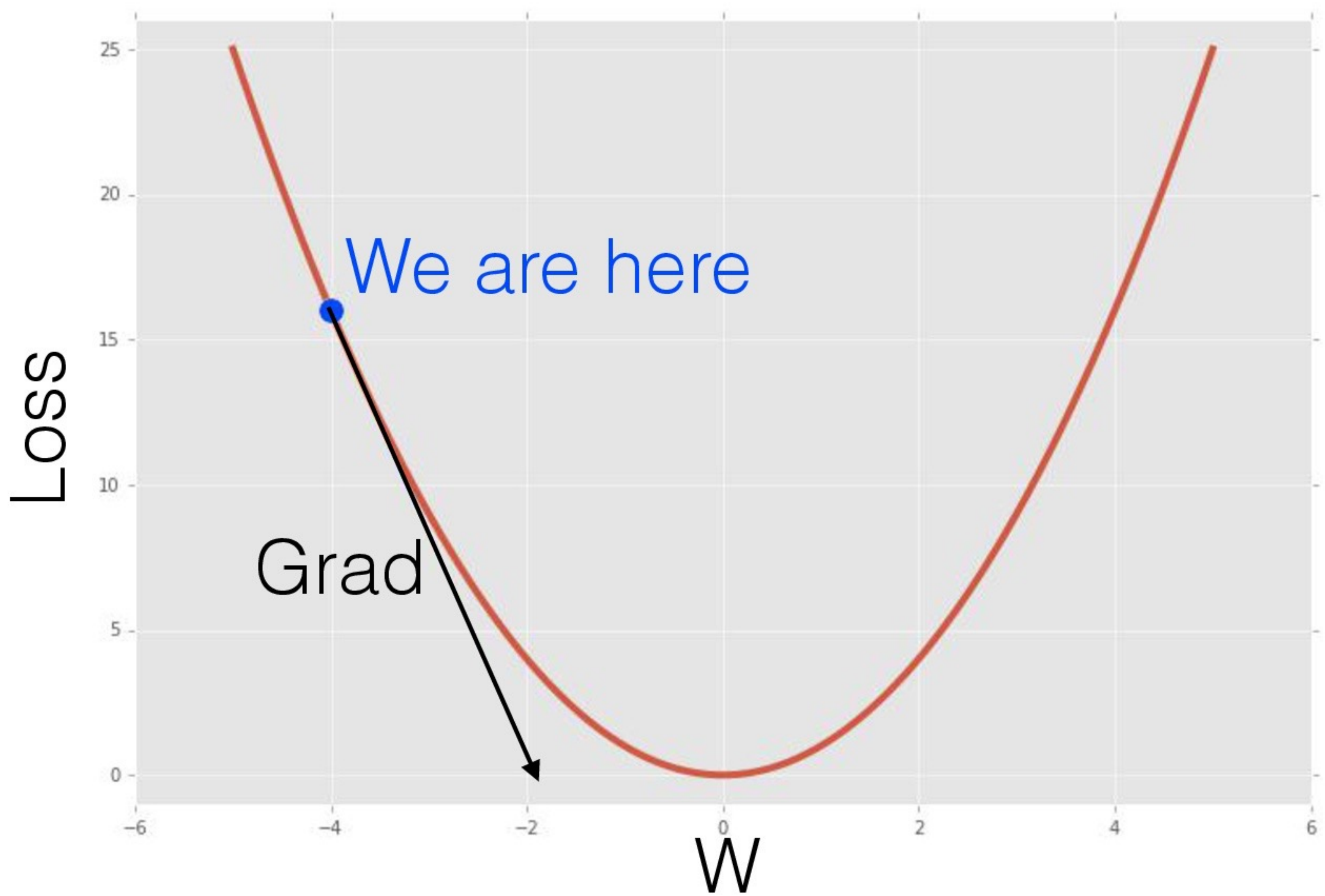
Epoch 1



Gradient Descent

$$W_t = W_{t-1} - \alpha \nabla W_{t-1}$$

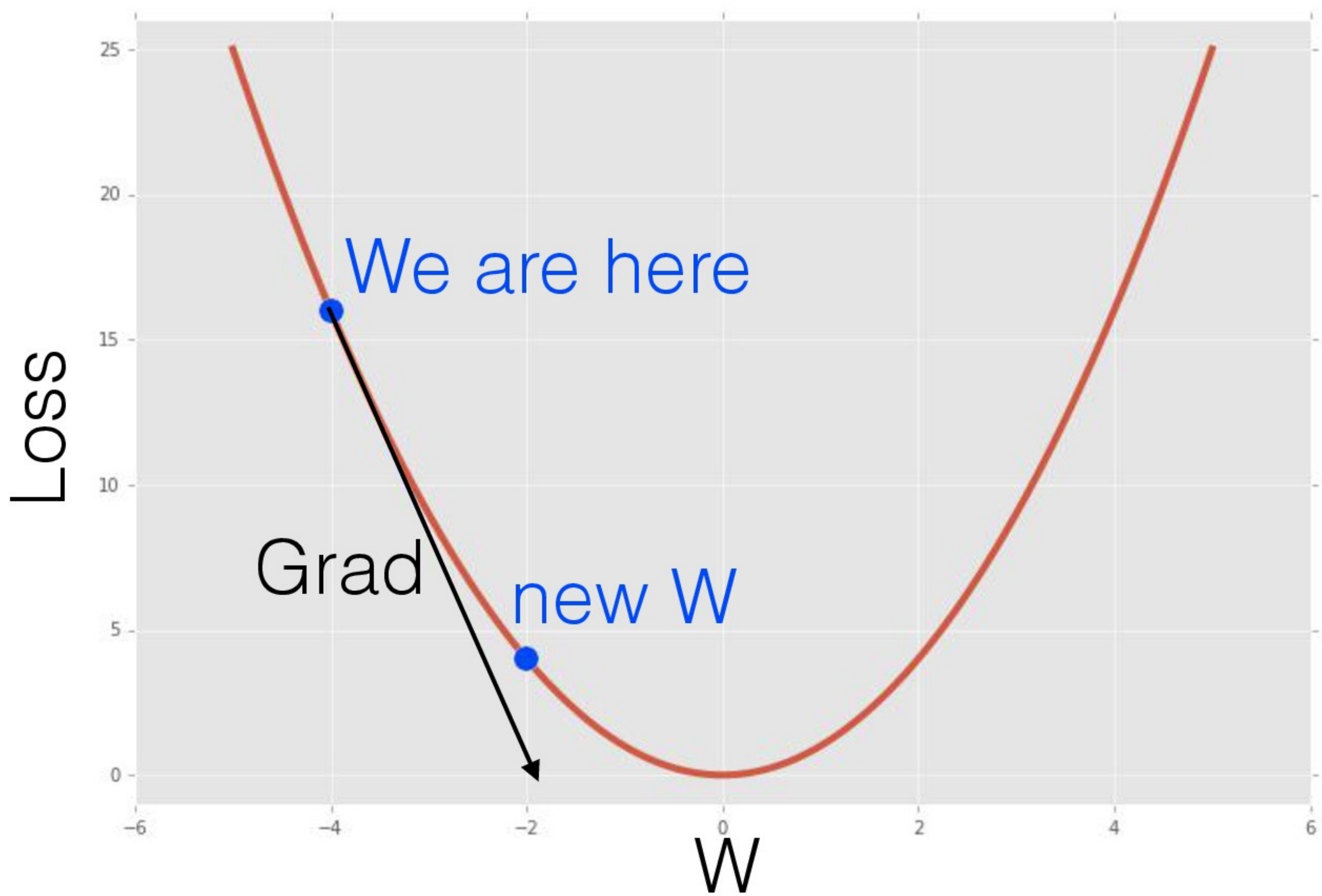
Epoch 1



Gradient Descent

$$W_t = W_{t-1} - \alpha \nabla W_{t-1}$$

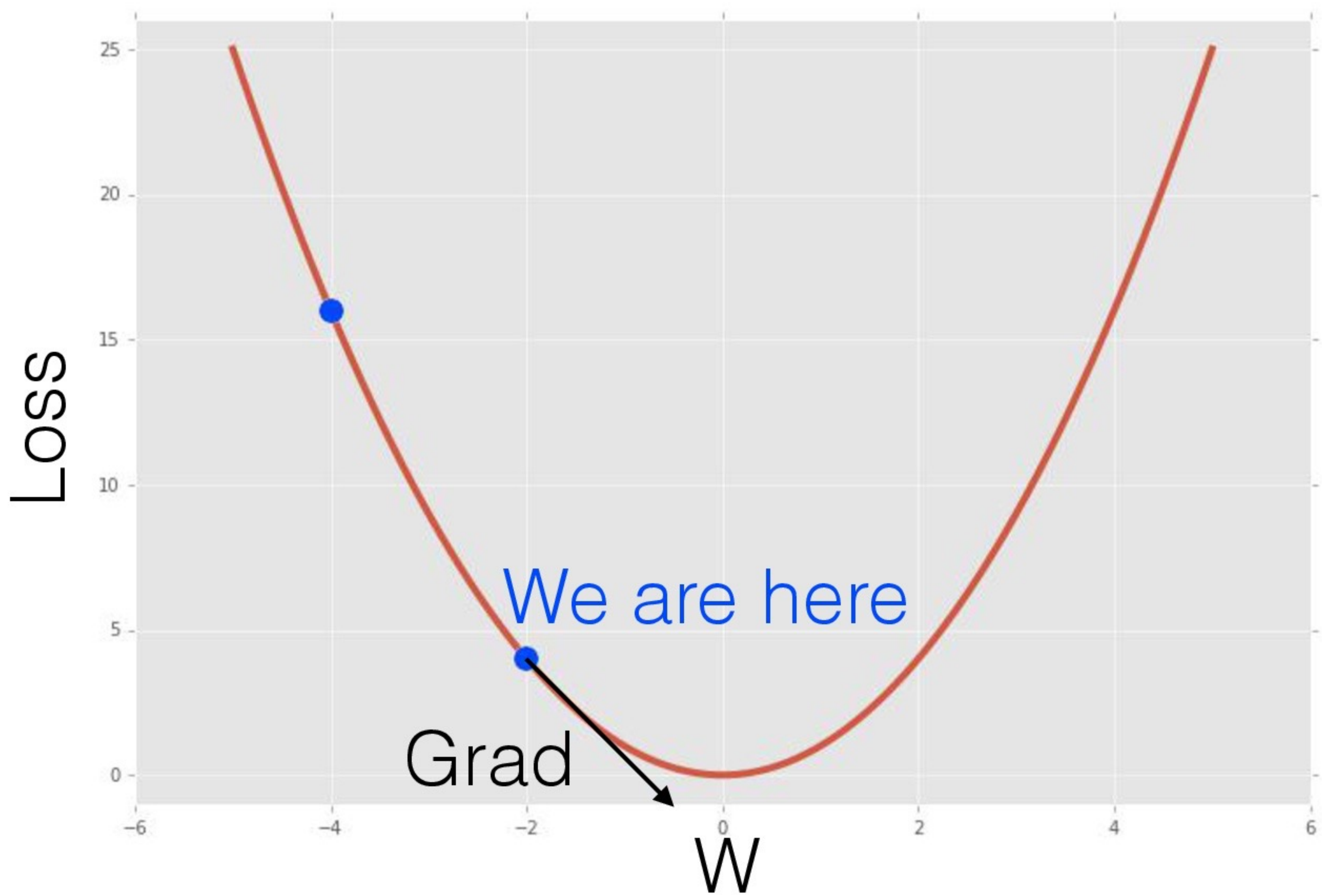
Epoch 2



Gradient Descent

$$W_t = W_{t-1} - \alpha \nabla W_{t-1}$$

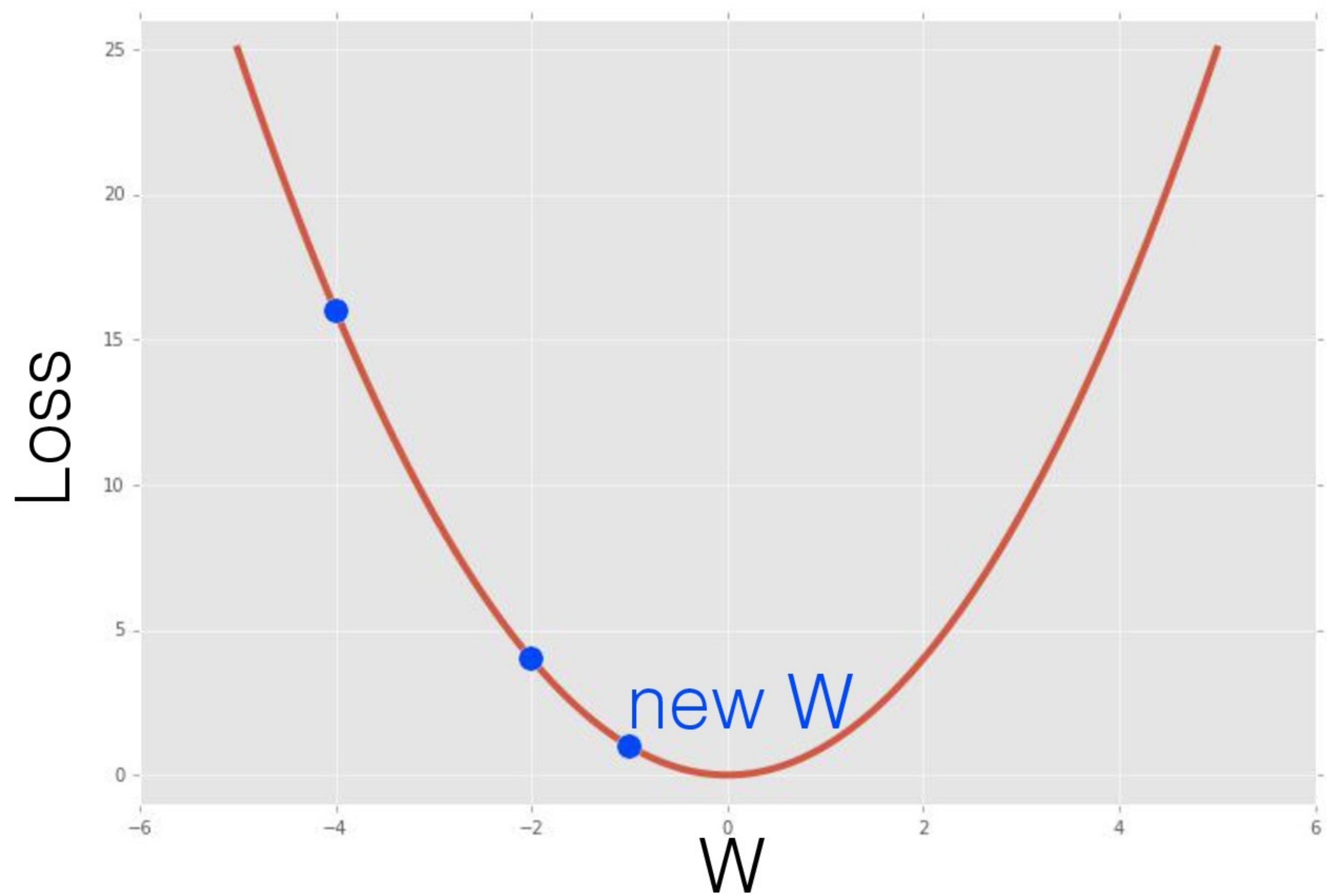
Epoch 2



Gradient Descent

$$W_t = W_{t-1} - \alpha \nabla W_{t-1}$$

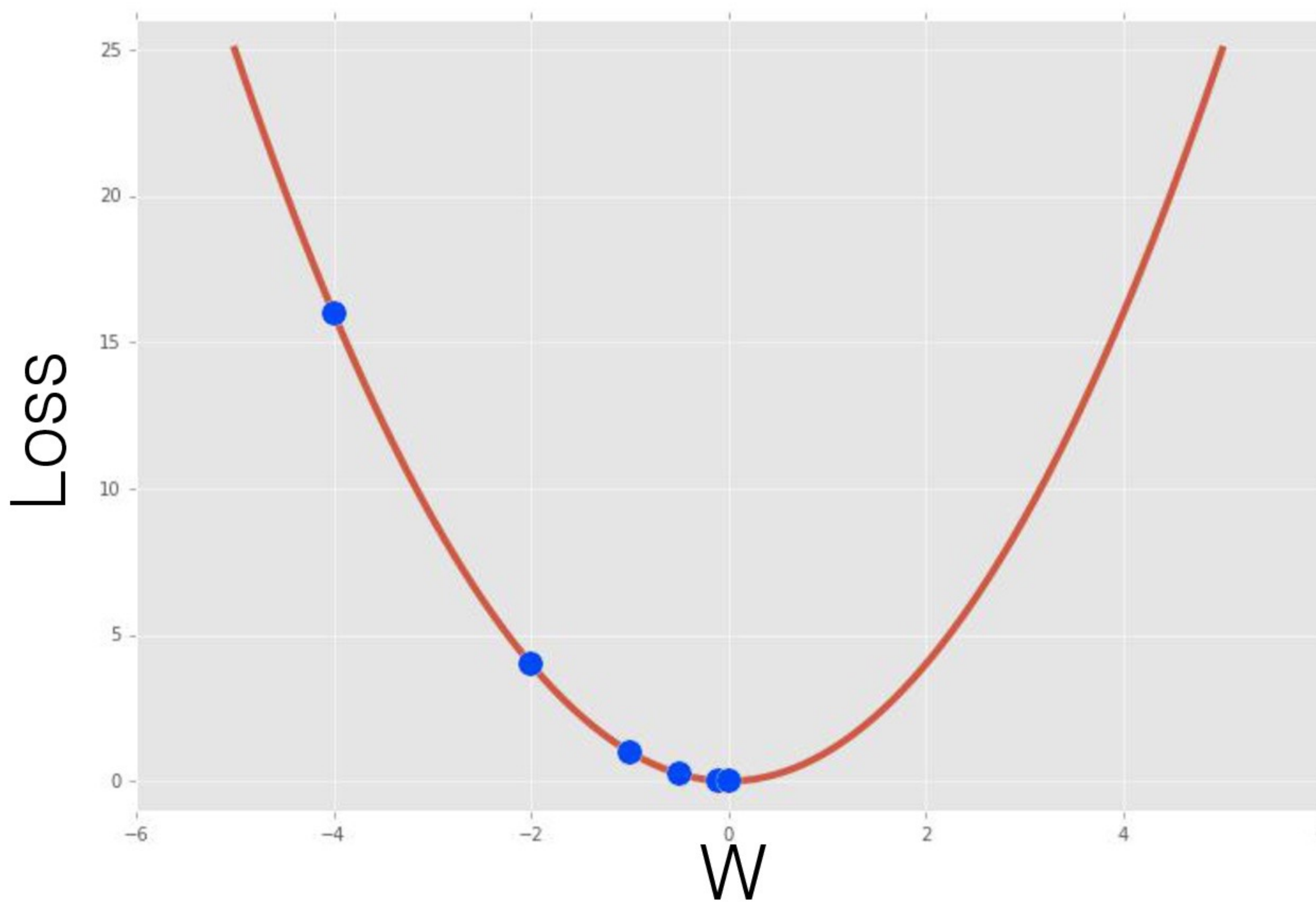
Epoch 3



Gradient Descent

$$W_t = W_{t-1} - \alpha \nabla W_{t-1}$$

Epoch K



Pros

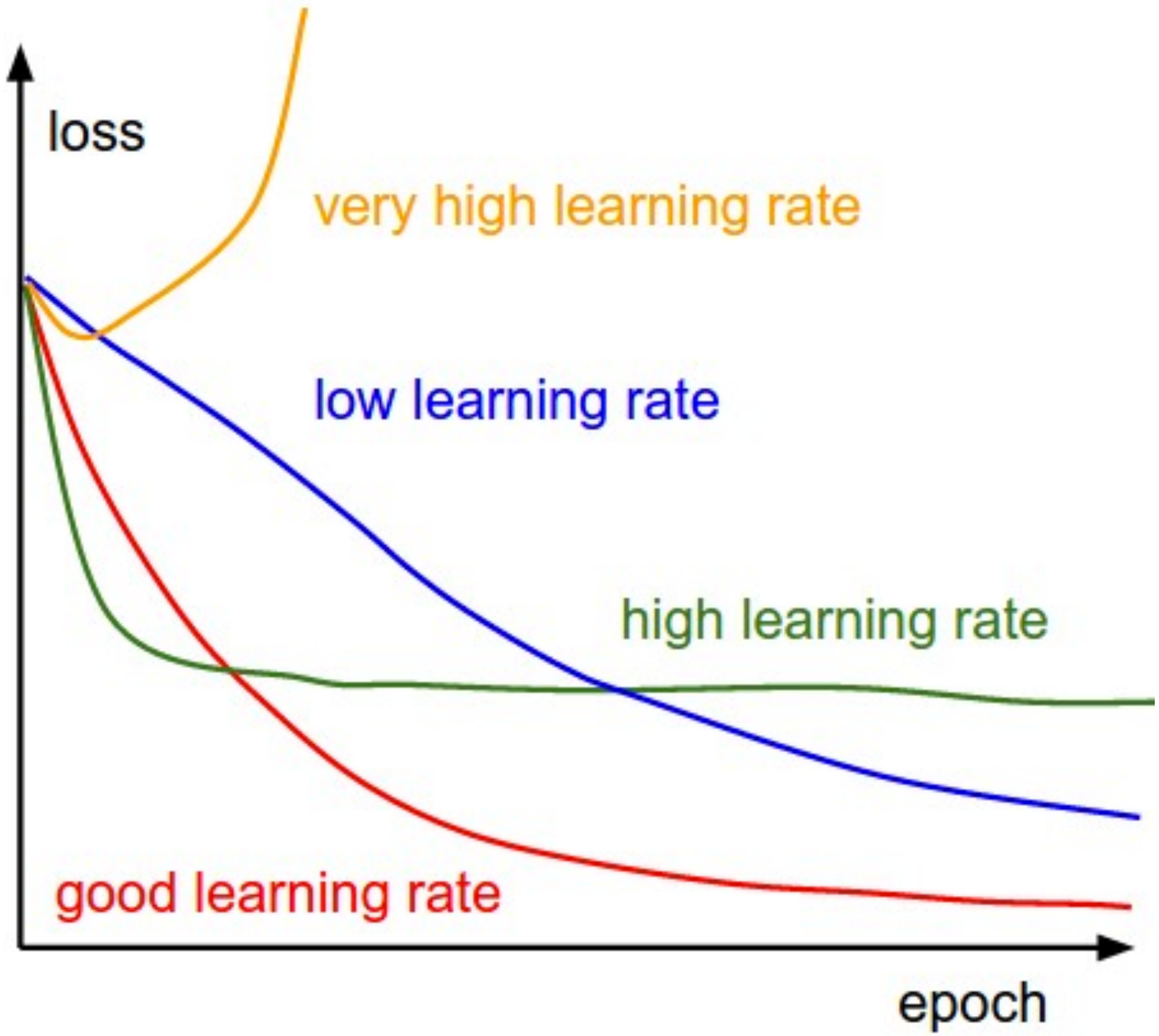
- If you can find a derivative, you can optimize*
- Every step in the correct direction
- Faster than explicit solution

Cons

Cons

- How to select learning rate?
- When to stop?
- Can’t determine global or local minimum
- Very slow on the ill-conditioned problems
- Difficult to compute on the big dataset
- No guarantees about finding minimum in finite time

Learning Rate



*can’t into derivatives

- Can’t explicitly find **df/dw_i** ?
- Compute it!
- **$df/dw_i \approx [f(w_i + eps) - f(w_i)]/eps \approx [f(w_i) - f(w_i - eps)]/eps \approx [f(w_i + eps) - f(w_i - eps)]/eps/2$**

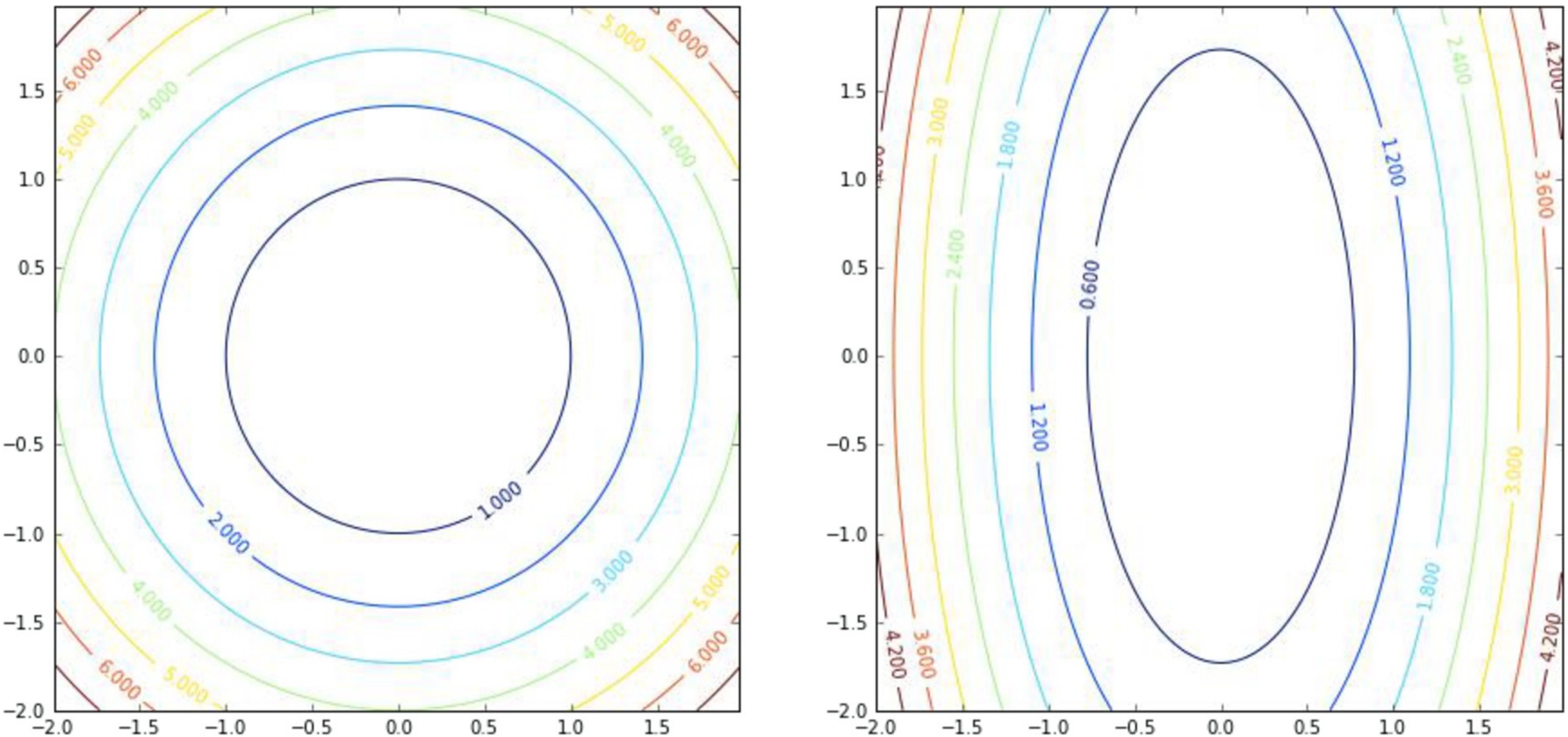
Check yourself!

- When implementing GD always check your explicit gradient function numerically!

- Calculate relative difference $|df_c - df_e|/\max(|df_e|, |df_c|)$

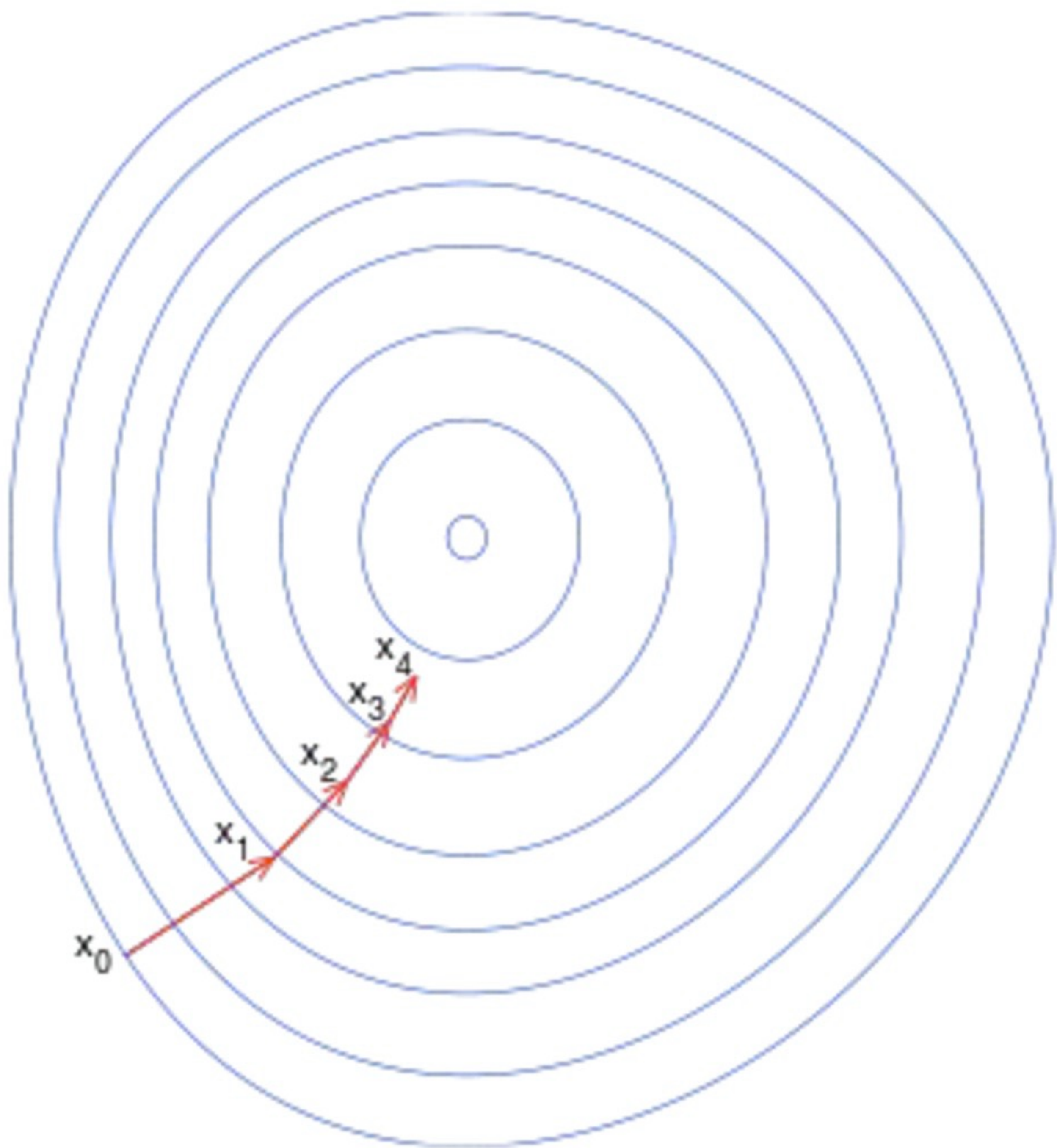
- relative error > 1e-2 usually means the gradient is probably wrong
- 1e-2 > relative error > 1e-4 should make you feel uncomfortable.
- 1e-7 and less you should be happy.
- eps ~ 1e-5
- Use float64 !

Standardization



Which one is better for SGD (GD)?

2d - Gradient Descent



Stochastic
Gradient Descent

Stochastic Gradient Descent

a.k.a. SGD

With GD you must pass through
the whole dataset to calculate one gradient!

X_1	X_2	X_3	X_4	X_5
				1
				1
				1
				1
				1
				1
				1
				1
				1
				1
				1
				1

Batch 1

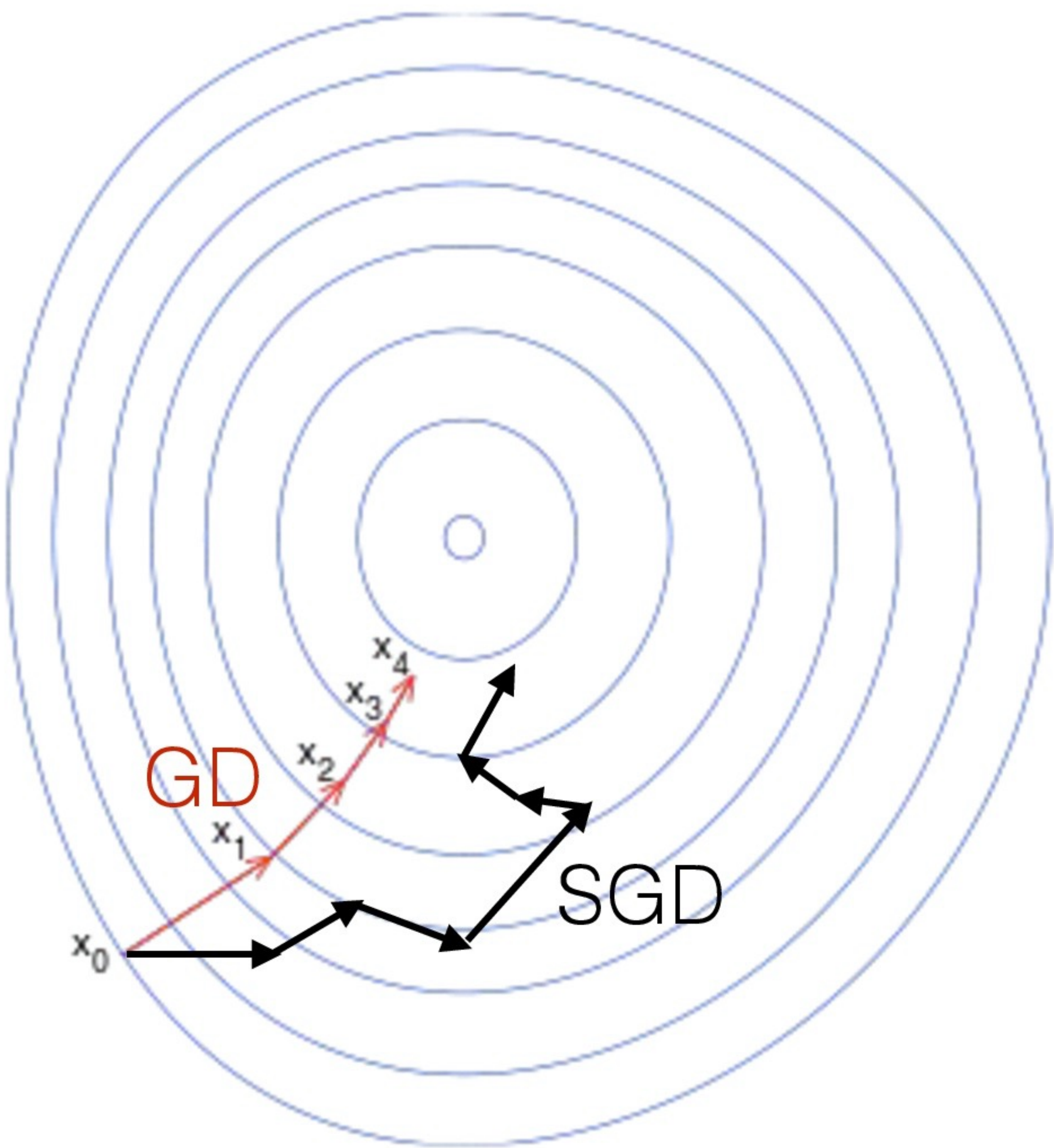
Batch 2

Batch 3

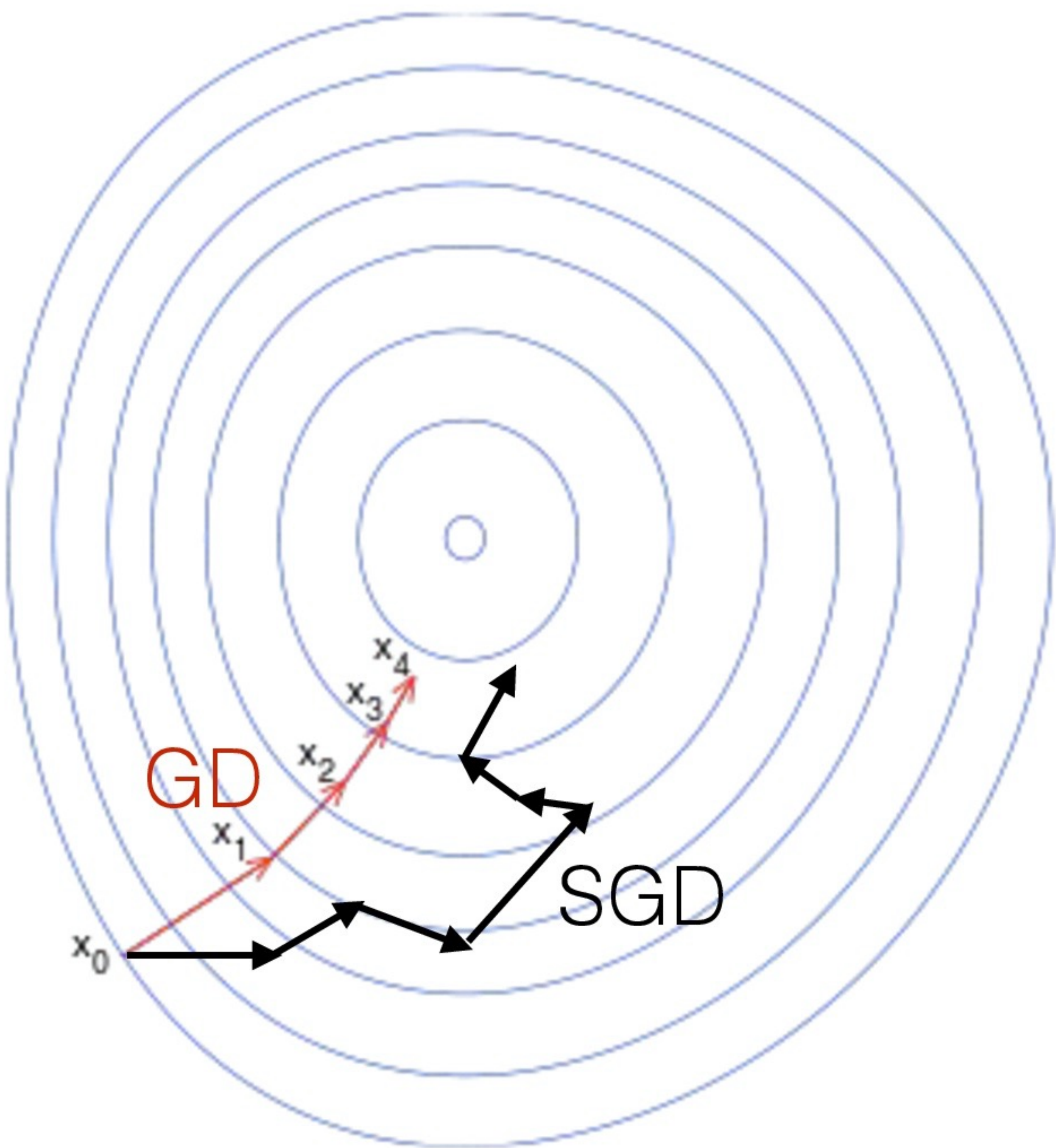
- Calculate batch gradient
- Update weights
- Repeat

That's one epoch!

GD vs SGD

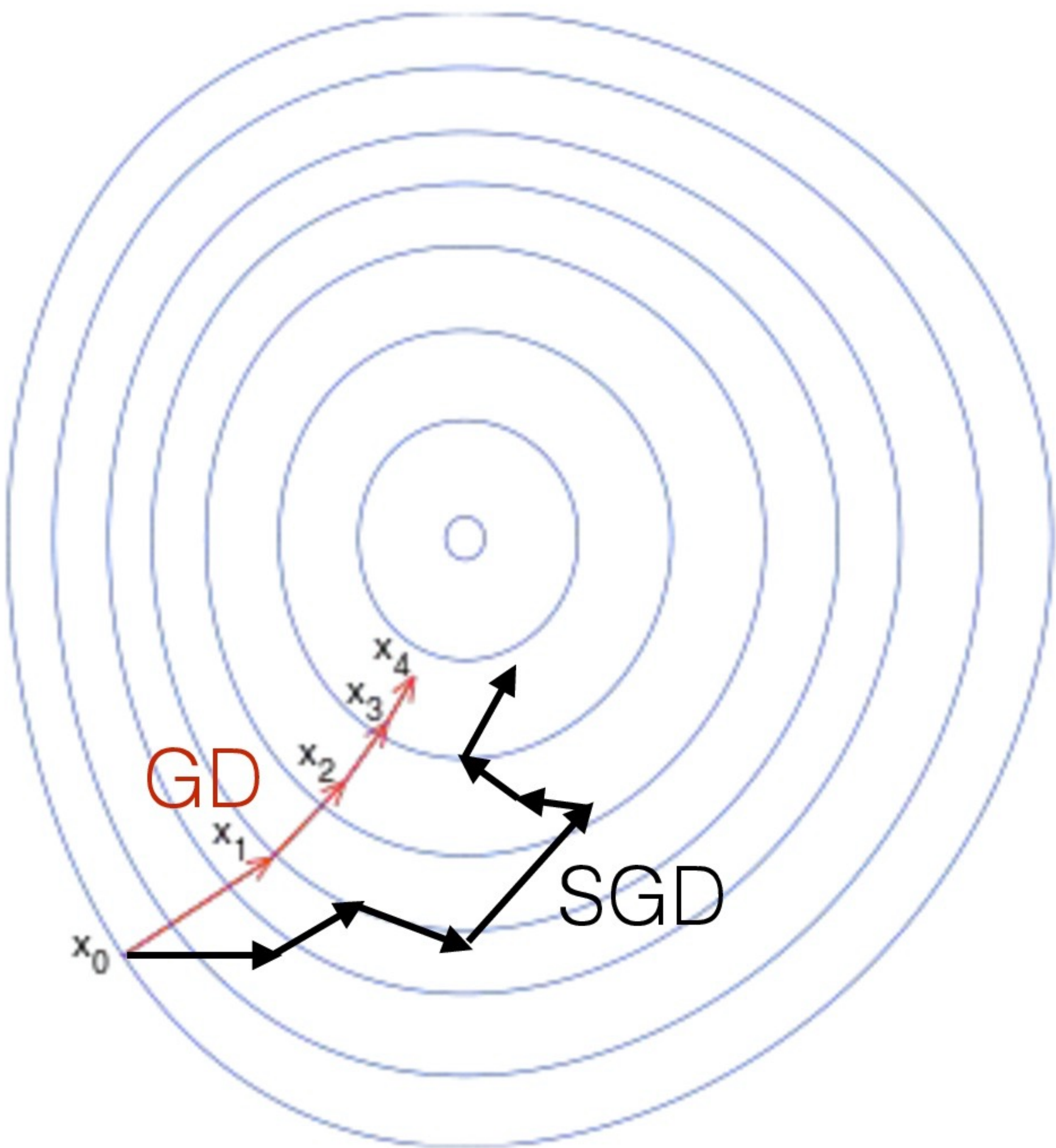


GD vs SGD



- (as GD) No guarantee about global minimum
- (as GD) No guarantee that solution would be found in finite time
- (as GD) No guarantee about convergence at all
- No guarantee about moving in correct direction

GD vs SGD



- (as GD) No guarantee about global minimum
- (as GD) No guarantee that solution would be found in finite time
- (as GD) No guarantee about convergence at all
- No guarantee about moving in correct direction

GD: O(n)
SGD: O(1)

Confusing names

- Full data: Gradient descent
- Part of the data: SGD, Mini batch GD
- One point: Fully stochastic GD, SGD

Newton

Moving in the right direction

$$\mathbf{x}_{n+1} = \mathbf{x}_n - [\mathbf{H}f(\mathbf{x}_n)]^{-1}\nabla f(\mathbf{x}_n)$$

- Super accurate

https://en.wikipedia.org/wiki/Newton%27s_method_in_optimization

Moving in the right direction

$$\mathbf{x}_{n+1} = \mathbf{x}_n - [\mathbf{H}f(\mathbf{x}_n)]^{-1} \nabla f(\mathbf{x}_n)$$

- Super accurate
- Need to compute Hessian (second order derivatives) **$O(n^2)$**
- Need to invert the Hessian! **$O(n^3)$**
- Never used in practice

https://en.wikipedia.org/wiki/Newton%27s_method_in_optimization

Momentum

Use momentum

Add momentum to your SGD path

$$\nabla W_t = \nabla W_t + \lambda \nabla W_{t-1}$$

Use ~0.9 momentum rate

Nesterov accelerated gradien (NAG)

Nesterov acceleration

$$\nabla W_t = \nabla(W_t + \lambda \nabla W_{t-1}) + \lambda \nabla W_{t-1}$$

Point: Nesterov is better than everything

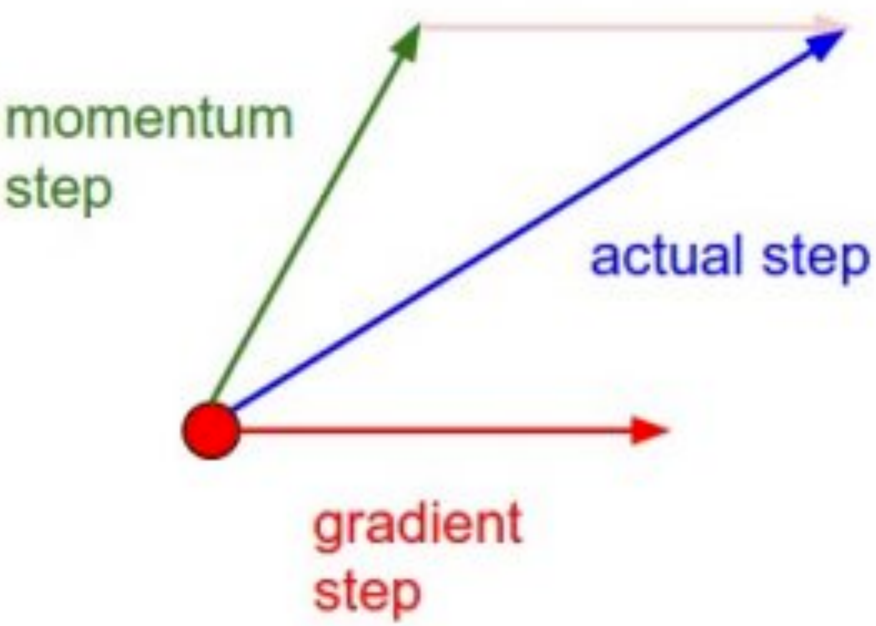
Use momentum

Add momentum to your SGD path

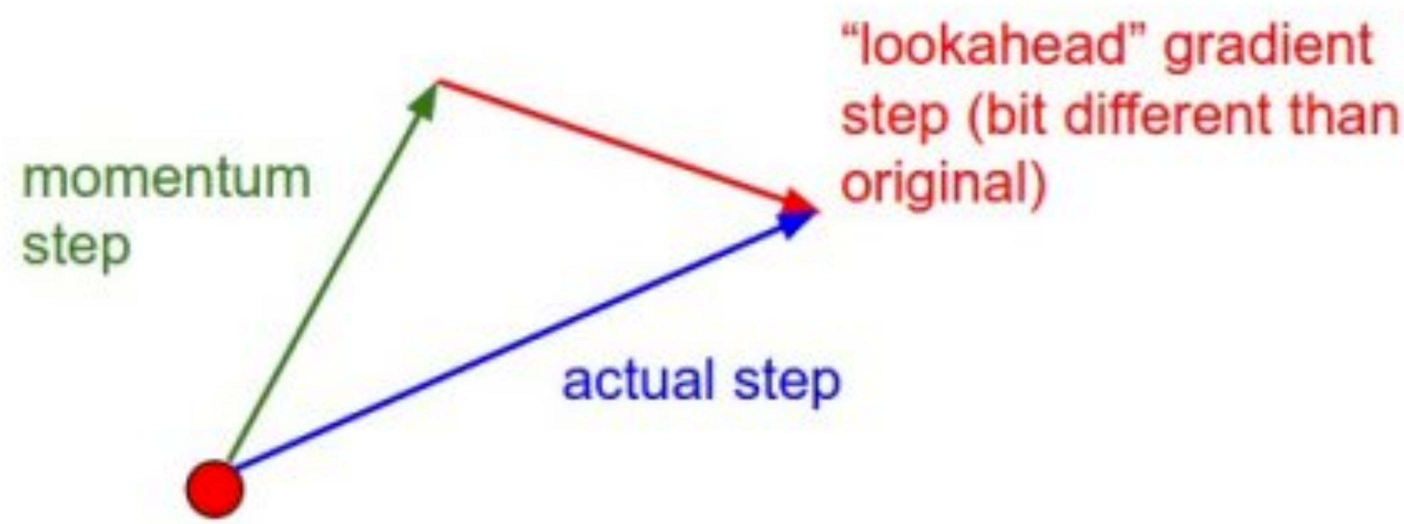
$$\nabla W_t = \nabla W_t + \lambda \nabla W_{t-1}$$

Use ~0.9 momentum rate

Momentum update



Nesterov momentum update



Idea: Slowing down the learning rate

It's good to reduce learning rate

- **Step decay.** Reduce the learning rate by some factor every few epochs. Typical values might be reducing the learning rate by a half every 5 epochs, or by 0.1 every 20 epochs.

- **Exponential decay.** has the mathematical form $\mathbf{a=a_0e^{-kt}}$, where $\boldsymbol{\alpha_0}$, **k** are hyperparameters and t is the iteration number (but you can also use units of epochs).

- **1/t decay.** The mathematical form $\boldsymbol{\alpha=\alpha_0/(1+kt)}$ where $\mathbf{a_0}$, **k** are hyperparameters and t is the iteration number.

Idea: Slowing down the learning rate with respect to parameter

AdaGrad

$$W_t = W_{t-1} - \alpha \frac{\nabla W_{t-1}}{\sqrt{G}}$$

Remember total update of every feature and scale updates to prevent jittering

AdaGrad

Adadelta

Adam

RMSprop

How to select?

- Try simple SGD.
- Add Momentum.
- Add Nesterov acceleration.
- Try some of adaptive methods

Resources

- http://cs231n.github.io/neural-networks-3/
- http://sebastianruder.com/optimizing-gradient-descent/
- https://tao.lri.fr/tiki-download_wiki_attachment.php?attId=954
- https://en.wikipedia.org/wiki/Numerical_differentiation