Machine Learning
ITCS 5356

Gradient Descent

Least Mean Squares

Razvan C. Bunescu

Department of Computer Science @ CCI

razvan.bunescu@charlotte.edu

ML is Optimization

- Try to find the value for *w* that minimizes:

$$J(w) = \frac{1}{2}w^2 - 4w + 9$$

$$J(w) = \frac{1}{2}(w - 4)^2 + 1$$

- Set $\nabla J(w) = 0$

  $\Rightarrow w - 4 = 0$

  $\Rightarrow w = 4$

Machine Learning is Optimization

- Parametric ML involves minimizing an **objective function** $J(\mathbf{w})$:
  - Also called **cost function** or **loss function**.
  - Want to find $\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} J(\mathbf{w})$

- Numerical optimization procedure:
  1. Start with some guess for $\mathbf{w}^0$, set $\tau = 0$.
  2. Update $\mathbf{w}^\tau$ to $\mathbf{w}^{\tau+1}$ such that $J(\mathbf{w}^{\tau+1}) \leq J(\mathbf{w}^\tau)$.
  3. Increment $\tau = \tau + 1$.
  4. Repeat from 2 until J cannot be improved anymore.

Gradient-based Optimization

- How to update $\mathbf{w}^{\tau}$ to $\mathbf{w}^{\tau+1}$ such that $J(\mathbf{w}^{\tau+1}) \leq J(\mathbf{w}^{\tau})$?

- Move $\mathbf{w}$ in the direction of **steepest descent**:

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} + \eta\boldsymbol{\Delta}$$

  - $\boldsymbol{\Delta}$ is the direction of steepest descent, i.e. direction along which J decreases the most.
  - η is the learning rate and controls the magnitude of the change.

Gradient-based Optimization

- Move $\mathbf{w}$ in the direction of **steepest descent**:

  $$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} + \eta\boldsymbol{\Delta}$$

- What is the direction of **steepest descent** of $J(\mathbf{w})$ at $\mathbf{w}^{\tau}$?
  - The gradient $\nabla J(\mathbf{w})$ is in the direction of **steepest ascent**.
  - Set $\boldsymbol{\Delta} = -\nabla J(\mathbf{w})$ => the **gradient descent** update:

  $$\boxed{\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta\nabla J(\mathbf{w}^{\tau})}$$

Gradient Descent Algorithm

- Want to minimize a function $J : R^n \rightarrow R$.
  - J is differentiable and convex.
  - compute gradient of J i.e. *direction of steepest increase*:

$$\nabla J(\mathbf{w}) = \left[\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \ldots, \frac{\partial J}{\partial w_n}\right]$$

1. Set learning rate $\eta = 0.001$ (or other small value).
2. Start with some guess for $\mathbf{w}^0$, set $\tau = 0$.
3. Repeat for epochs E or until J does not improve:
4. $\quad \tau = \tau + 1$.
5. $\quad \mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta \nabla J(\mathbf{w}^{\tau})$

What if objective is not differentiable?

- **Subgradient methods.**
  - Minimize convex functions that are not necessarily differentiable.
- **Gradient free methods**:
  - **Evolutionary Programming**.
  - **Bayesian Optimization.**
    - https://arxiv.org/abs/1807.02811
  - **Particle swarm optimization.**
  - **Surrogate optimization**
  - **Simulated annealing.**
  - …

Gradient Descent Algorithm

- Want to minimize a function $J : R^n \rightarrow R$.
  - J is differentiable and convex.
  - compute gradient of J i.e. *direction of steepest increase*:

$$\nabla J(\mathbf{w}) = \left[\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \ldots, \frac{\partial J}{\partial w_n}\right]$$

1. Set learning rate $\eta = 0.001$ (or other small value).
2. Start with some guess for $\mathbf{w}^0$, set $\tau = 0$.
3. Repeat for epochs E or until J does not improve:
4. $\quad\tau = \tau + 1$.
5. $\quad\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta \nabla J(\mathbf{w}^{\tau})$

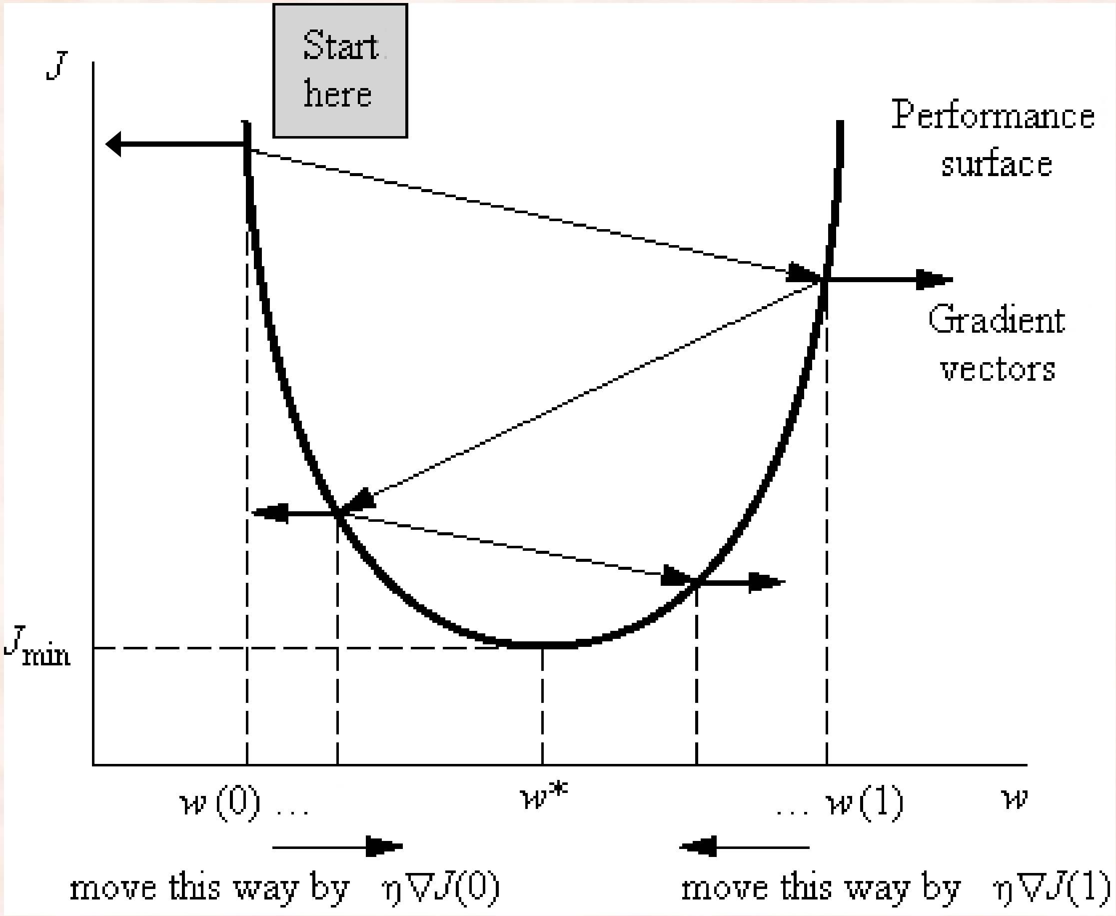
Gradient Descent: Example

- Let's use gradient descent to minimize:

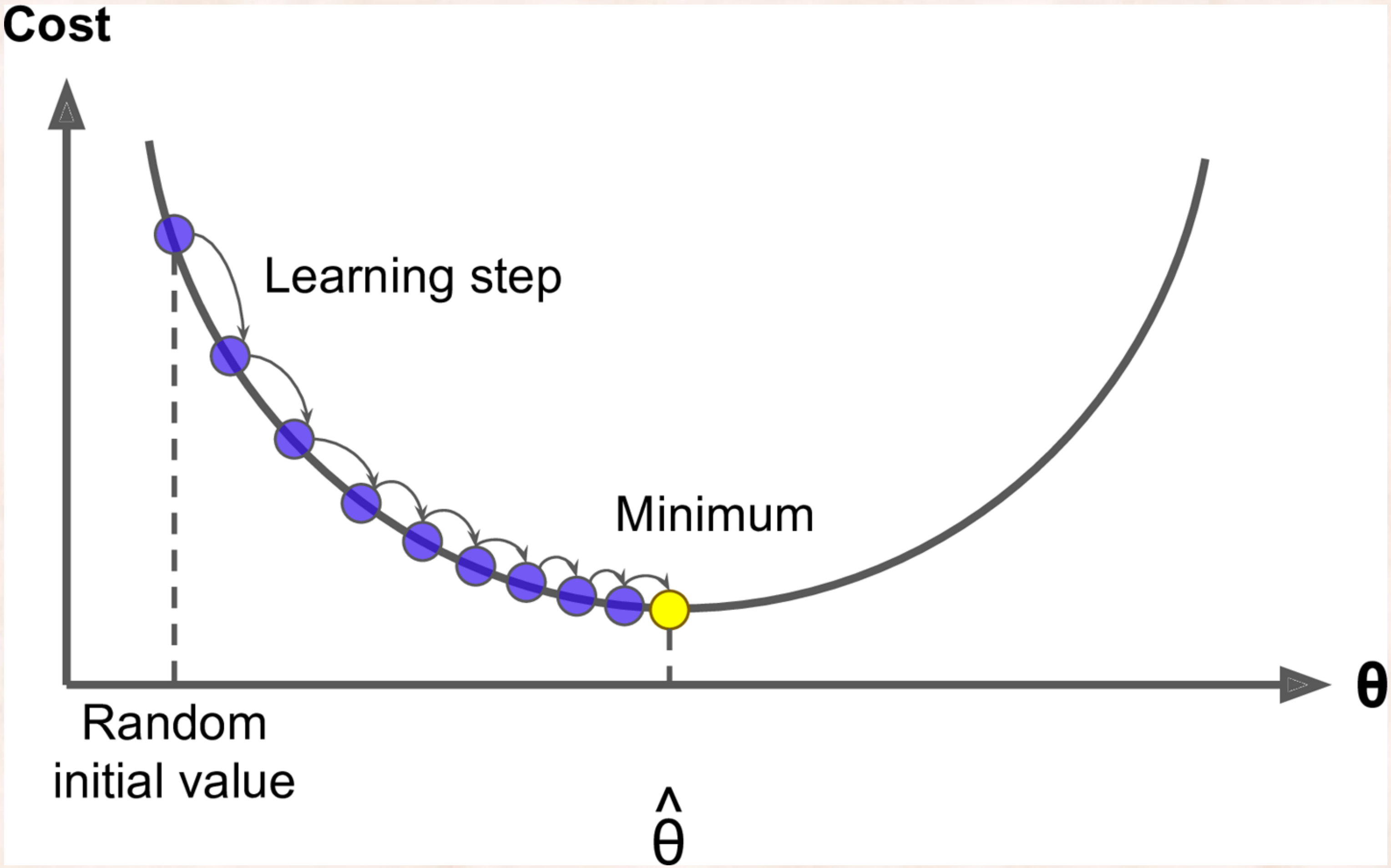
$$J(w) = \frac{1}{2}w^2 - 4w + 9$$

  - Start from $w^0 = 0$, use learning rate $\eta = 0.5$
  - What if $\eta = 1.0$?
  - What if $\eta = 2.0$?

Gradient Descent: Large Updates



Gradient Descent: Small Updates



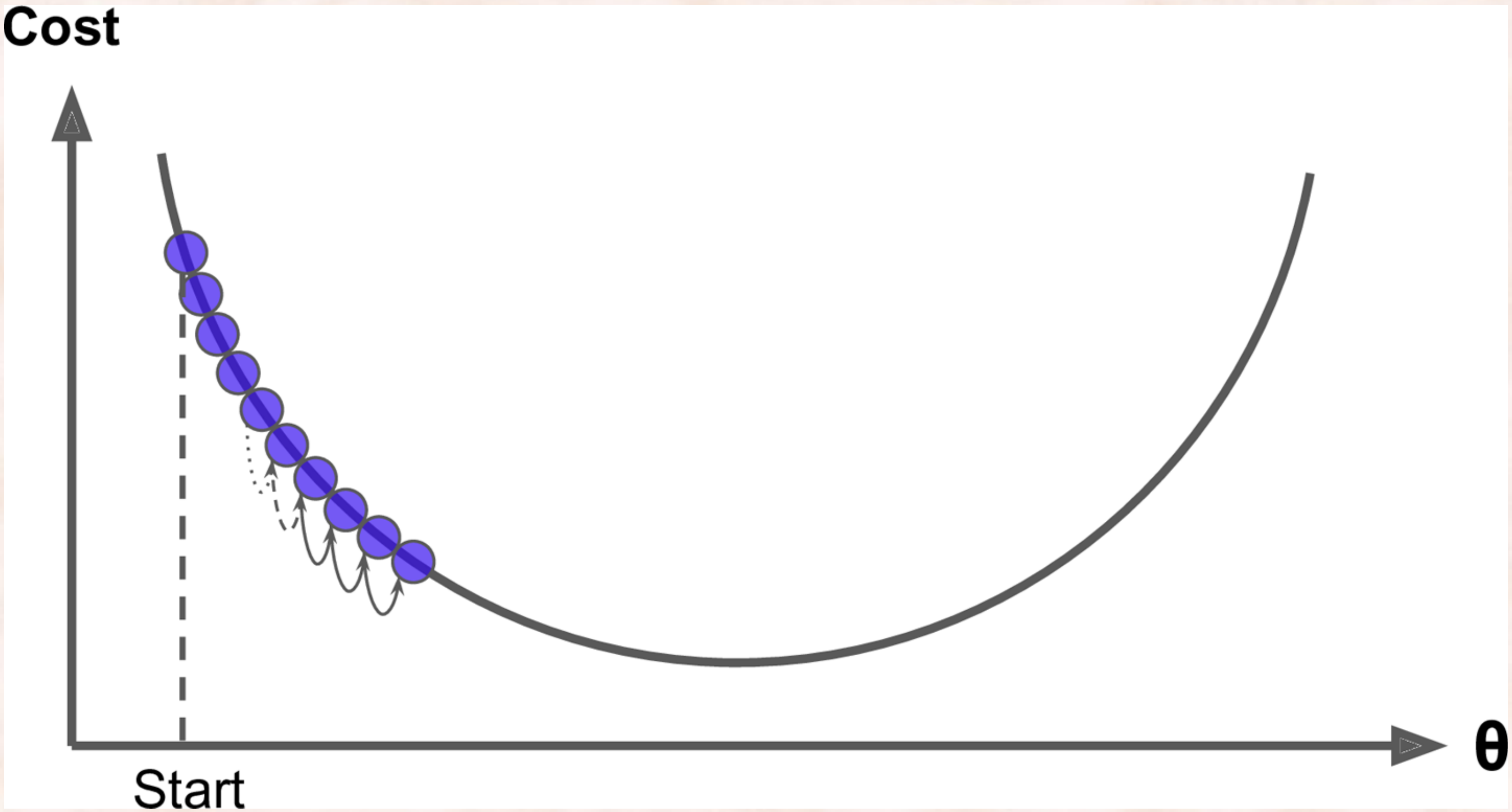
https://www.safaribooksonline.com/library/view/hands-on-machine-learning

The Learning Rate

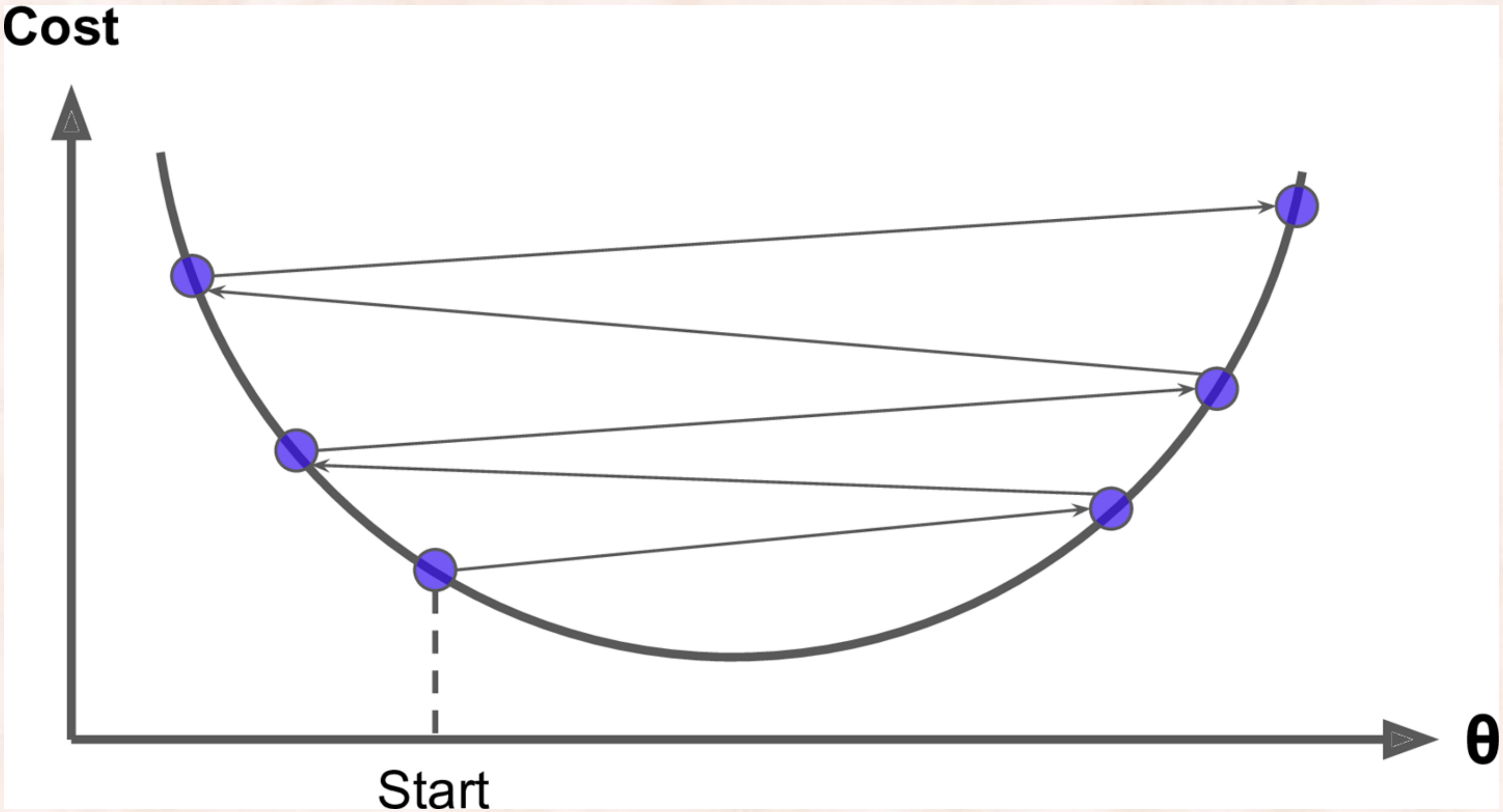
1. Set **learning rate** η = 0.001 (or other small value).
2. Start with some guess for $\mathbf{w}^0$, set $\tau = 0$.
3. Repeat for epochs E or until J does not improve:
4. $\tau = \tau + 1$.
5. $\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta \nabla J(\mathbf{w}^{\tau})$

- How big should the **learning rate** be?
  - If learning rate too small => slow convergence.
  - If learning rate too big => oscillating behavior => may not even converge.

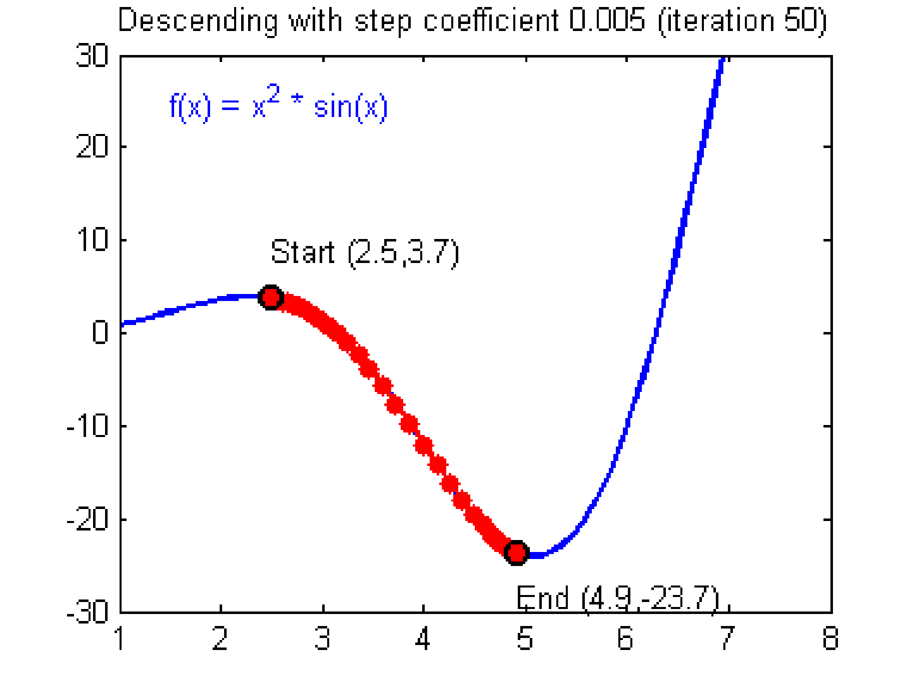
Learning Rate too Small

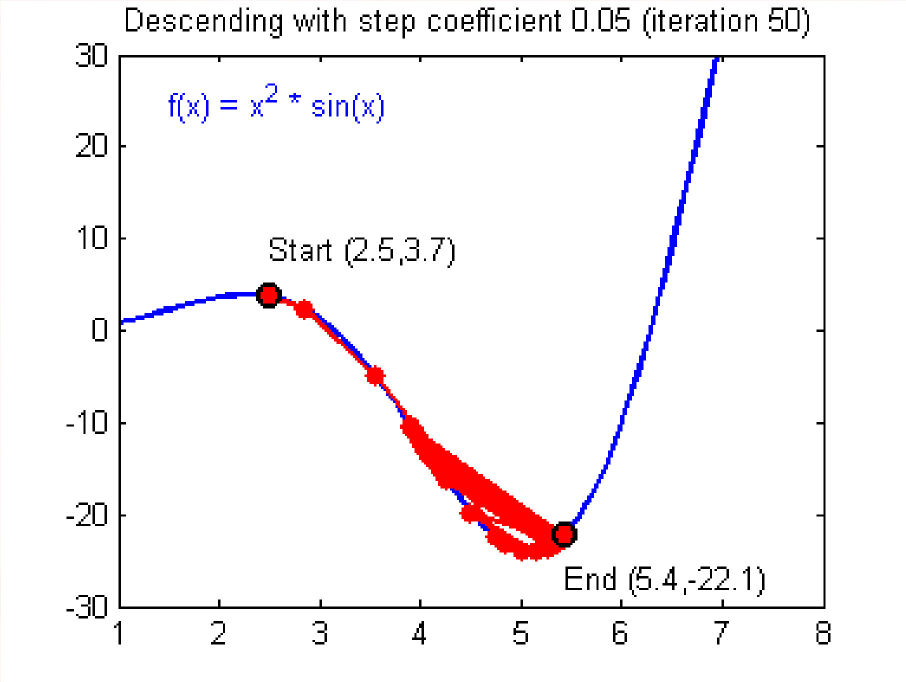


Learning Rate too Large



Learning Rates vs. GD Behavior





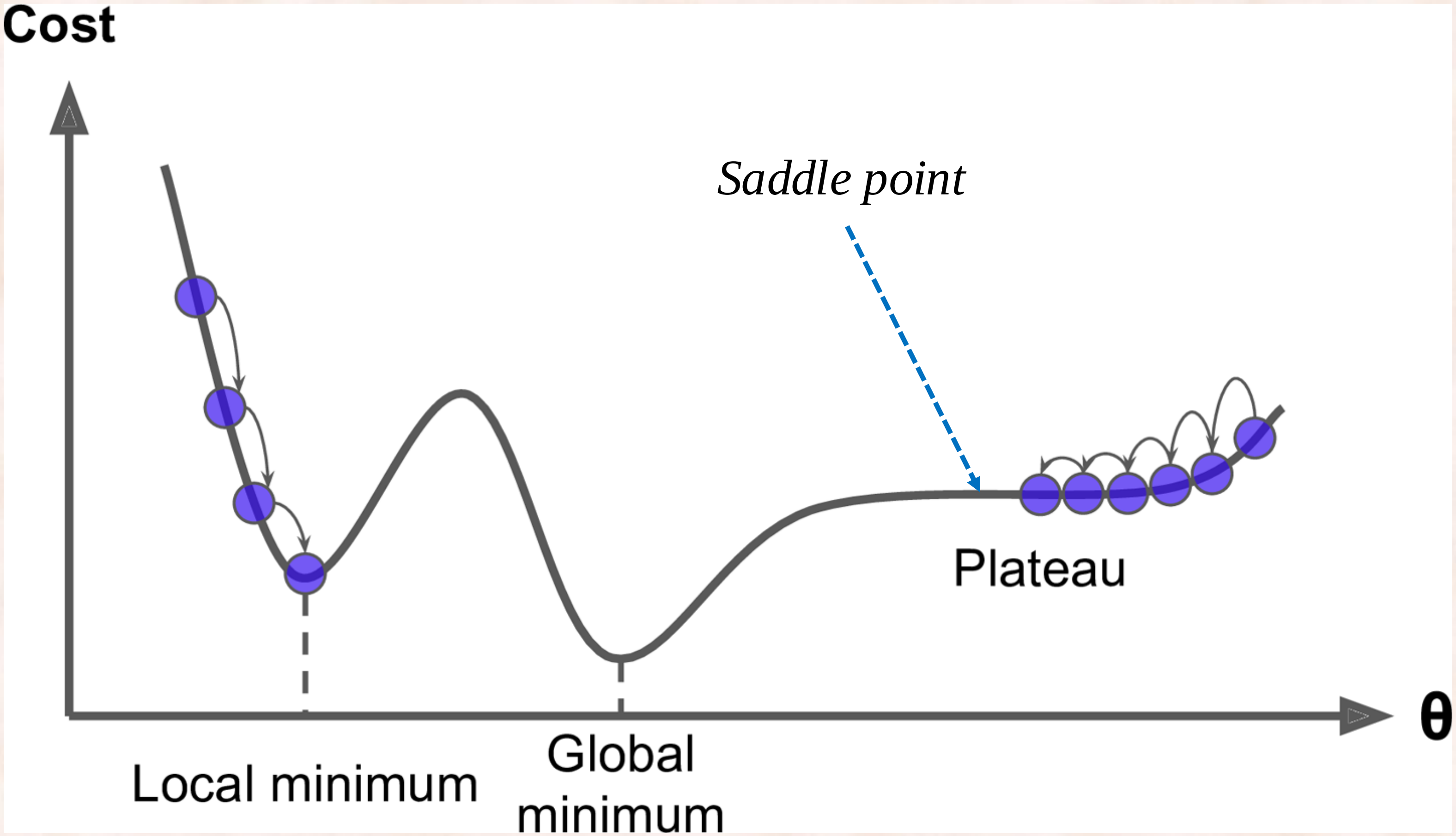
http://scs.ryerson.ca/~aharley/neural-networks/

The Learning Rate

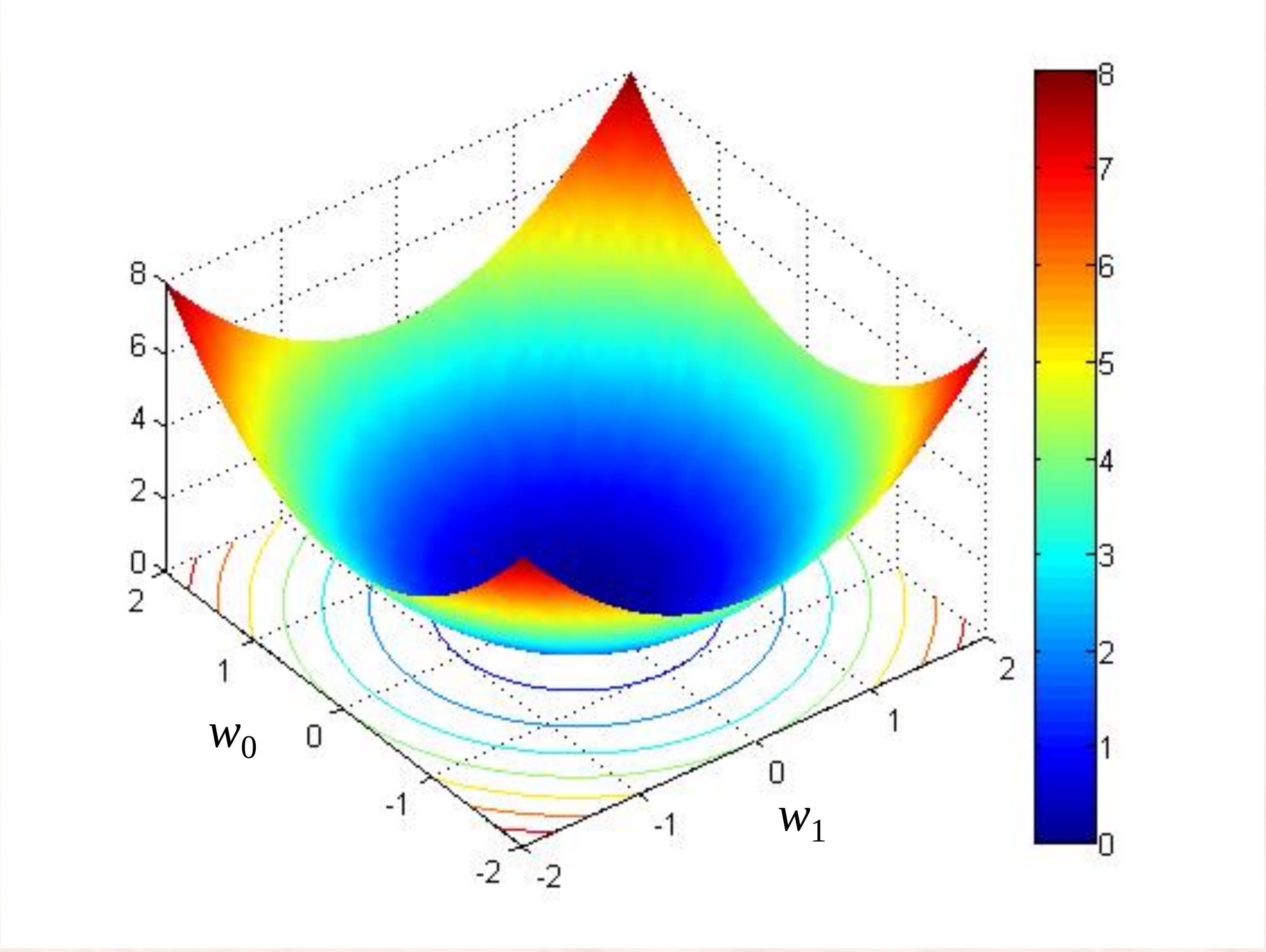
- How big should the **learning rate** be?
  - If learning rate too big => oscillating behavior.
  - If learning rate too small => hinders convergence.

- Use **line search** (backtracking line search, conjugate gradient, …).
- Use **second order methods** (Newton's method, L-BFGS, ...).
  - Requires computing or estimating the Hessian.
- Use a simple learning rate **annealing schedule**:
  - Start with a relatively large value for the learning rate.
  - Decrease the learning rate as a function of the number of epochs or as a function of the improvement in the objective.
- Use **adaptive learning rates**:
  - Adagrad, Adadelta, RMSProp, Adam.

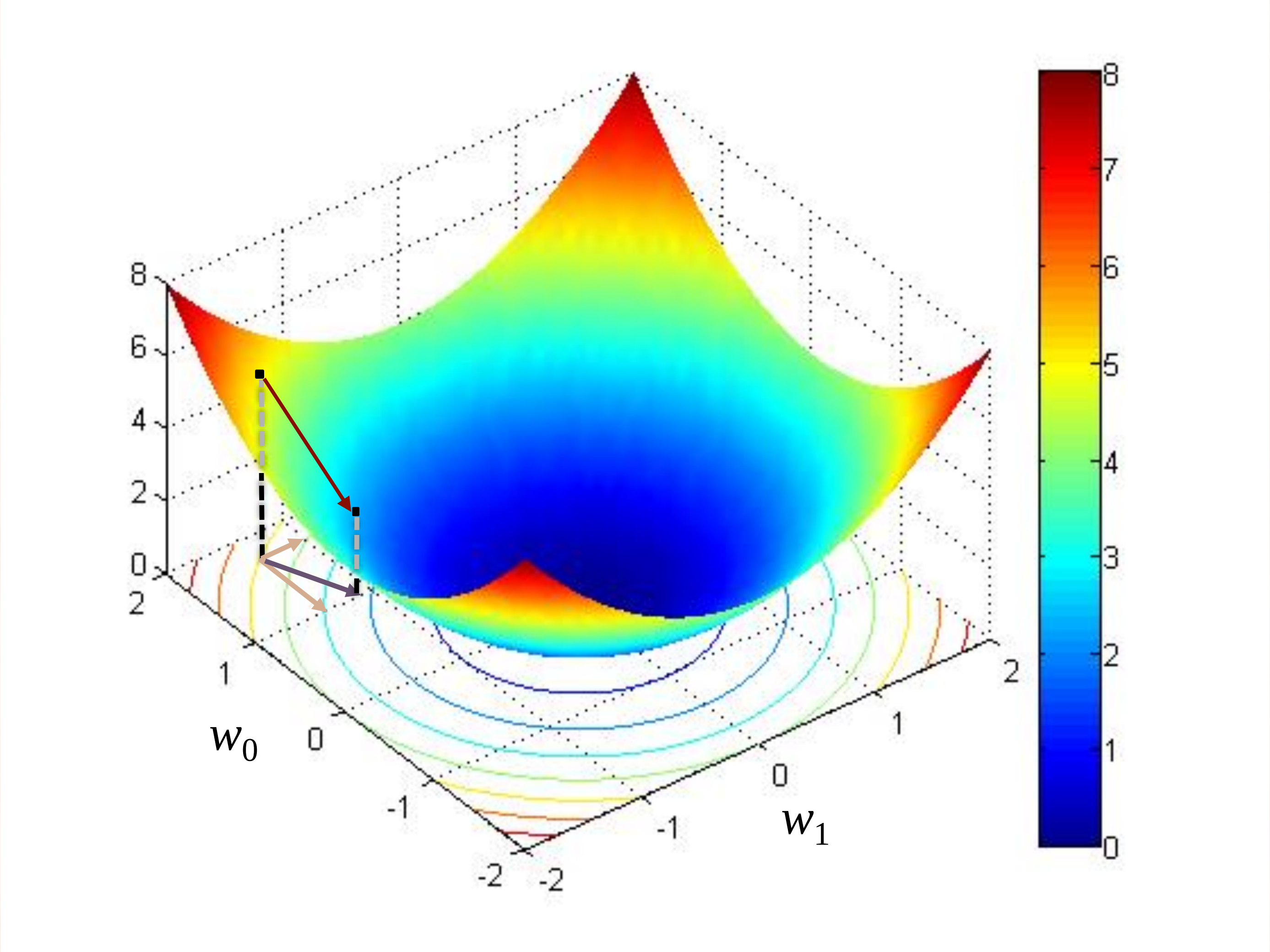
Gradient Descent: Nonconvex Objective



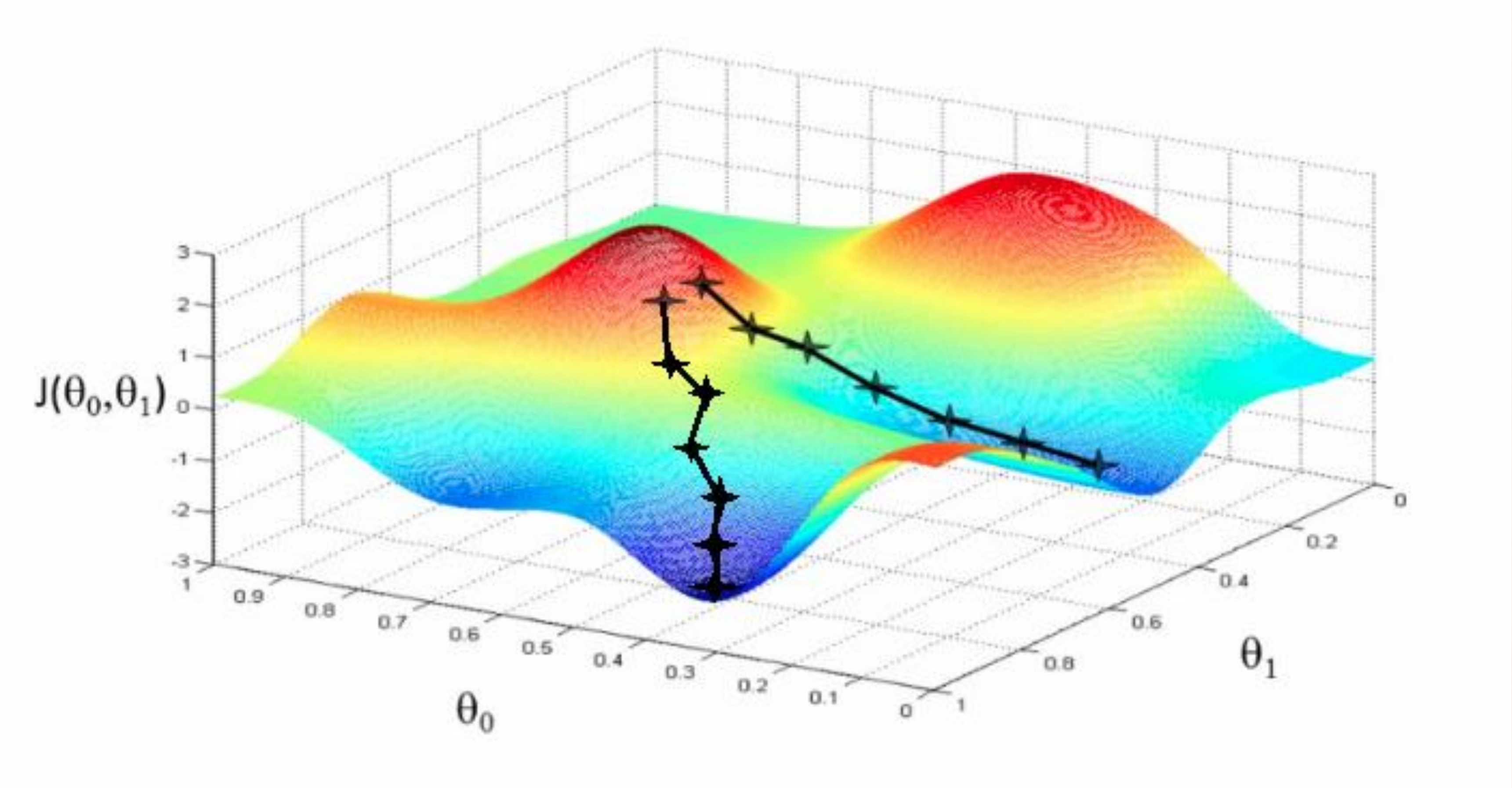
Convex Multivariate Objective



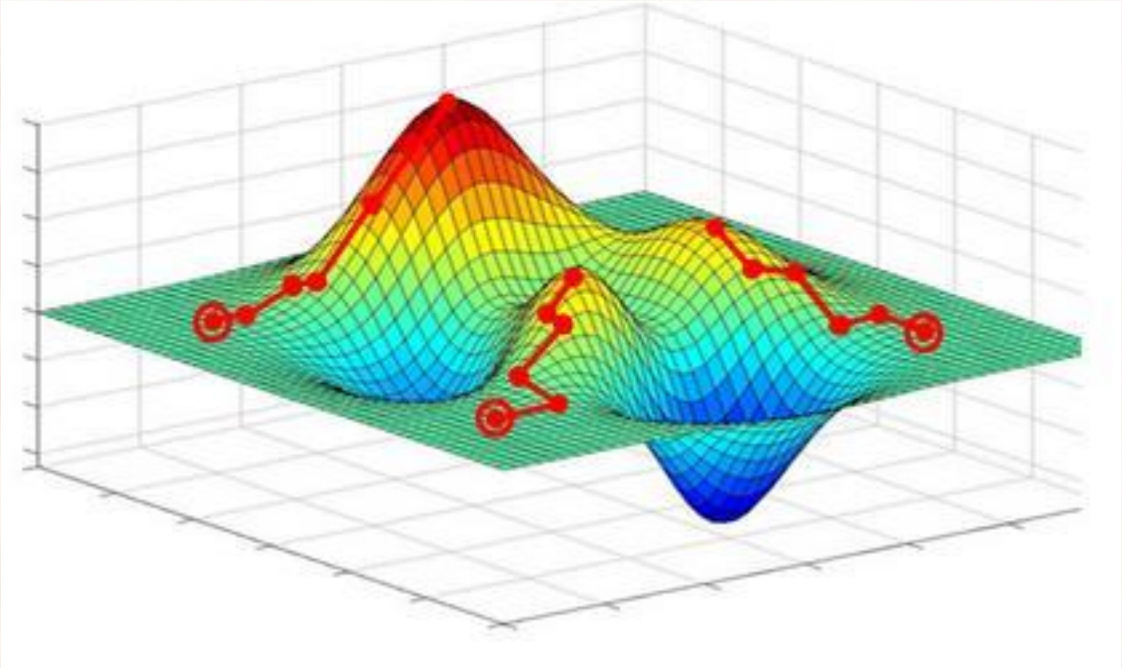
Gradient Step and Contour Lines



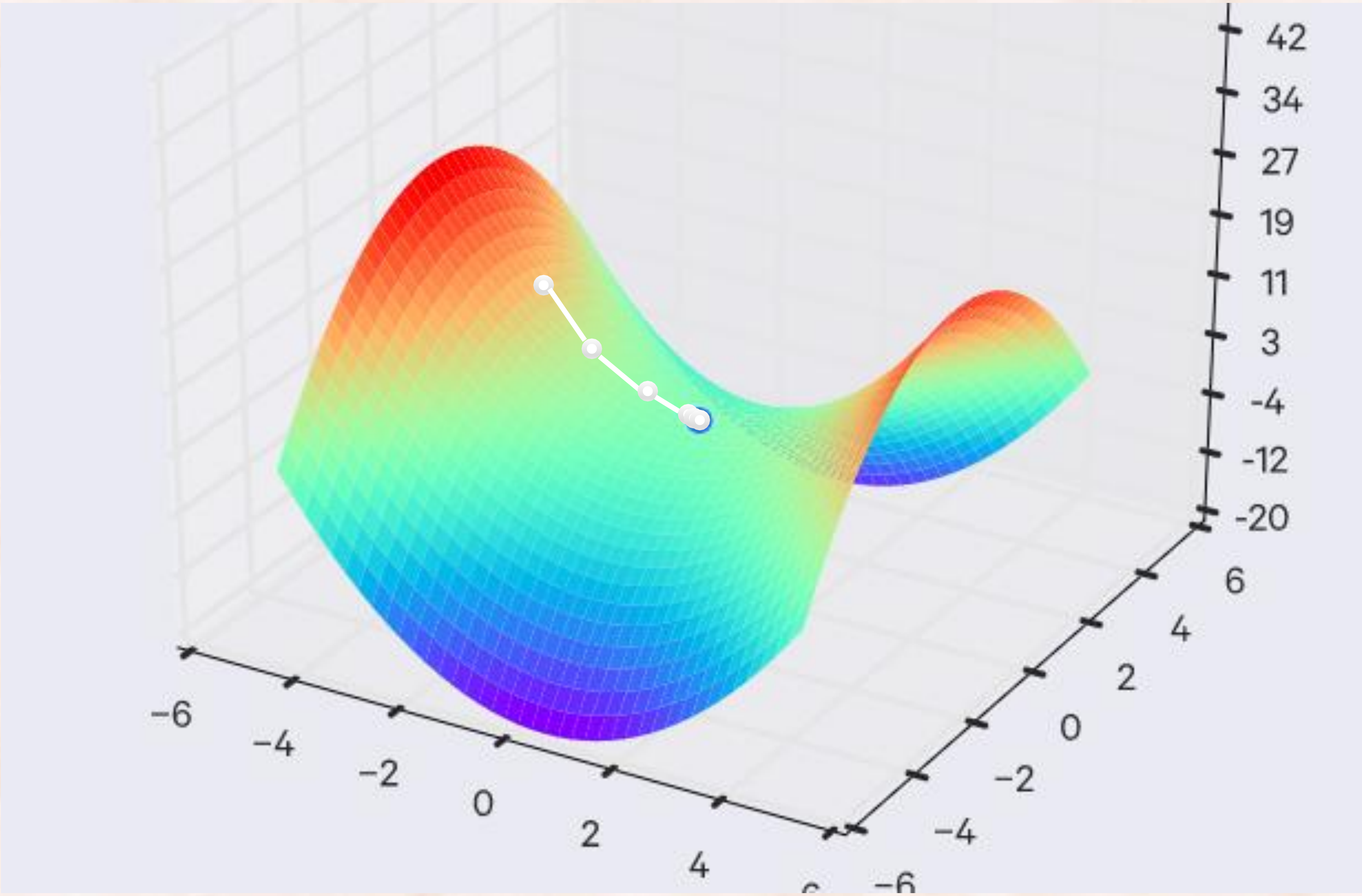
Gradient Descent: Nonconvex Objectives



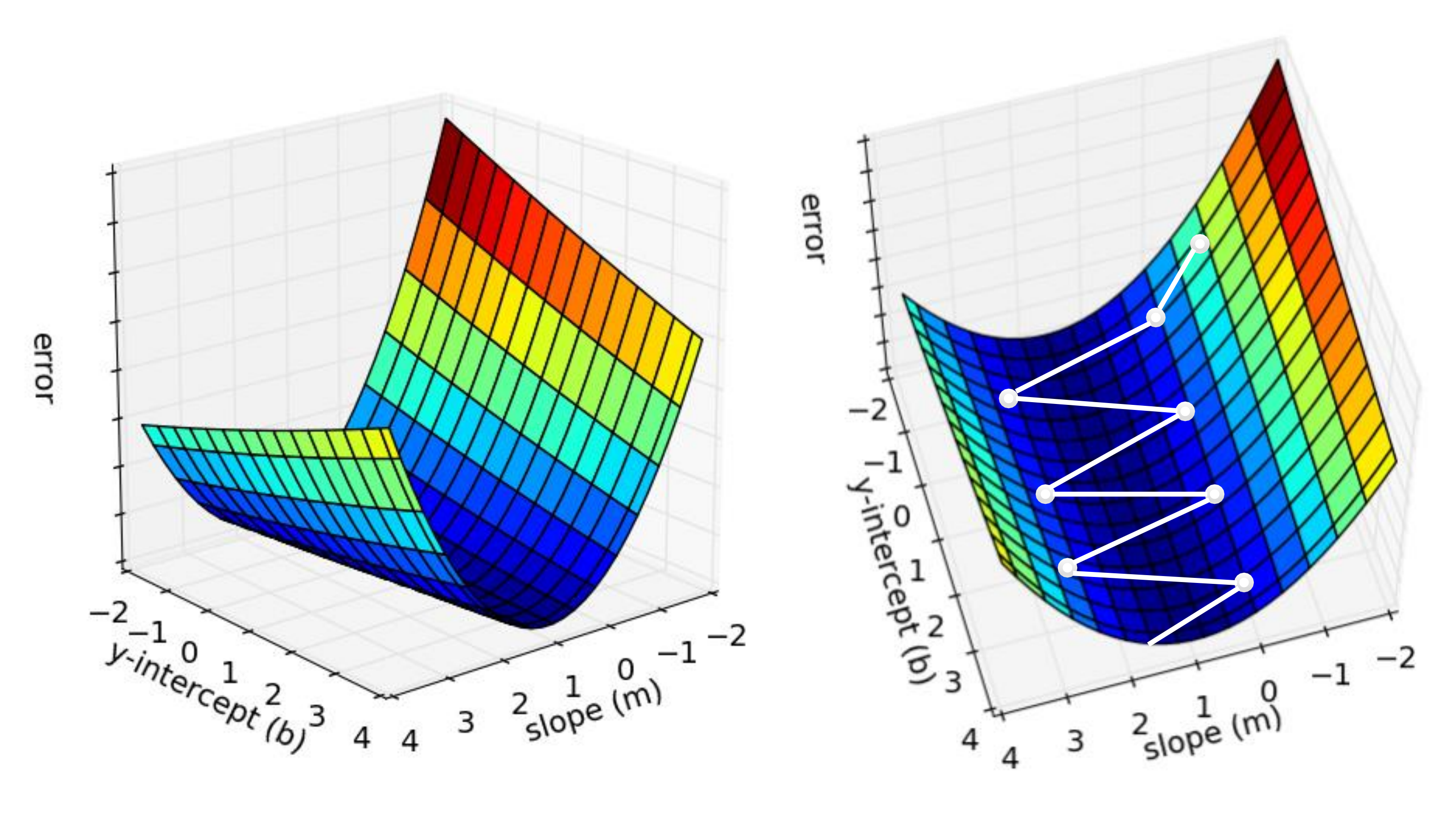
Gradient Descent & Plateaus



Gradient Descent & Saddle Points



Gradient Descent & Ravines



Gradient Descent & Ravines

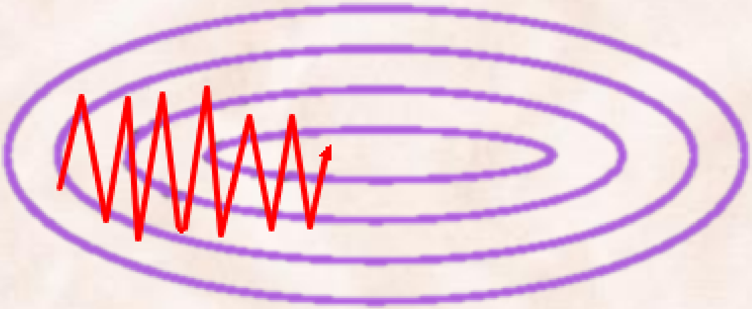
- **Ravines** are areas where the cost surface curves much more steeply in one dimension than another.
  - Common around local optima.
  - GD oscillates across the slopes of the ravines, making slow progress towards the local optimum along the bottom.
- Use **momentum** to help accelerate GD in the relevant directions and dampen oscillations:
  - Add a fraction of the past **update vector** to the current update vector.
    - The momentum term increases for dimensions whose previous gradients point in the same direction.
    - It reduces updates for dimensions whose gradients change sign.
    - Also <u>reduces the risk of getting stuck in local minima</u>.

Gradient Descent & Momentum

Vanilla Gradient Descent:

$$\mathbf{v}^{\tau+1} = \eta \nabla J(\mathbf{w}^{\tau})$$

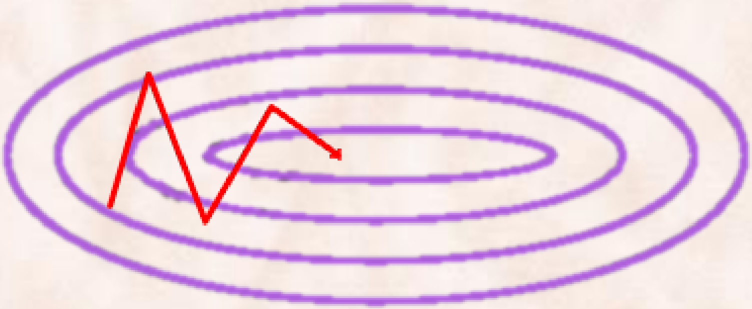
$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \mathbf{v}^{\tau+1}$$



Gradient Descent w/ Momentum:

$$\mathbf{v}^{\tau+1} = \gamma \mathbf{v}^{\tau} + \eta \nabla J(\mathbf{w}^{\tau})$$

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \mathbf{v}^{\tau+1}$$



γ is usually set to 0.9 or similar.

The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions.

Batch vs. Stochastic Gradient Descent

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta \boxed{\nabla J(\mathbf{w}^{\tau})}$$

- Depending on how much data is used to compute the gradient at each step:
  - **Batch gradient descent**:
    - Use all the training examples.
  - **Stochastic gradient descent** (SGD).
    - Use one training example, update after each.
    - **Minibatch gradient descent**.
      - Use a constant number of training examples (minibatch).

Batch Gradient Descent for Linear Regression

$\hat{y}_n = \mathbf{w}^T \mathbf{x}^{(n)}$

- Sum-of-squares error:

$$J(\mathbf{w}) = \frac{1}{2N} \sum_{n=1}^{N} \left(\mathbf{w}^T \mathbf{x}^{(n)} - y_n \right)^2$$

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta \, \nabla J(\mathbf{w}^{\tau})$$

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta \frac{1}{N} \sum_{n=1}^{N} \left(\mathbf{w}^{\tau^T} \mathbf{x}^{(n)} - t_n \right) \mathbf{x}^{(n)}$$

Stochastic Gradient Descent for Linear Regression

$\hat{y}_n = \mathbf{w}^T \mathbf{x}^{(n)}$

- Sum-of-squares error:

$$J(\mathbf{w}) = \frac{1}{2N} \sum_{n=1}^{N} \left(\mathbf{w}^T \mathbf{x}^{(n)} - y_n\right)^2 = \frac{1}{N} \sum_{n=1}^{N} loss\left(\mathbf{w}^\tau, \mathbf{x}^{(n)}\right)$$

$$\mathbf{w}^{\tau+1} = \mathbf{w}^\tau - \eta \, \nabla loss\left(\mathbf{w}^\tau, \mathbf{x}^{(n)}\right)$$

$$\boxed{\mathbf{w}^{\tau+1} = \mathbf{w}^\tau - \eta\left(\mathbf{w}^T \mathbf{x}^{(n)} - y_n\right) \mathbf{x}^{(n)}}$$

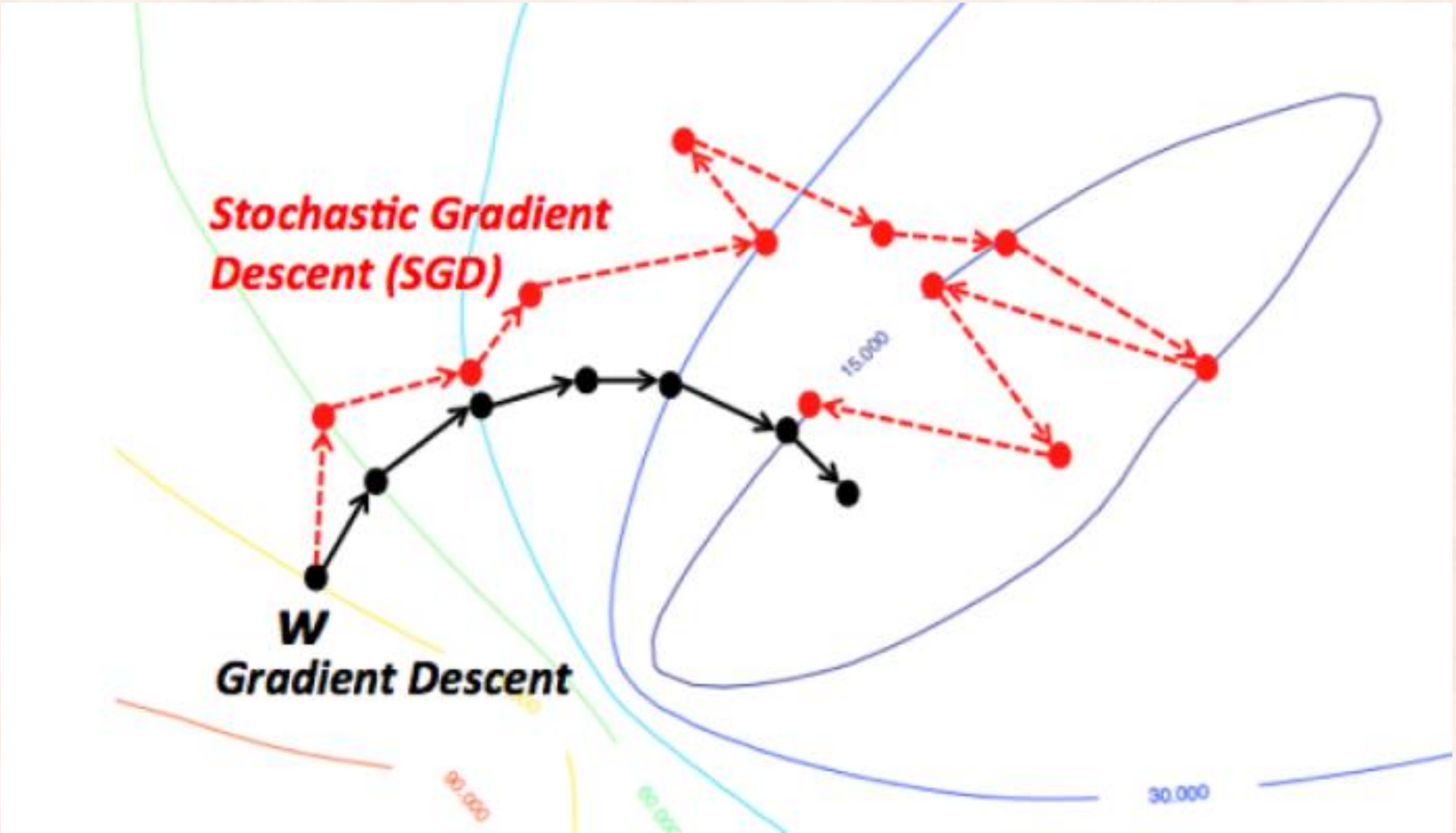
- Update parameters **w** after each example, sequentially:

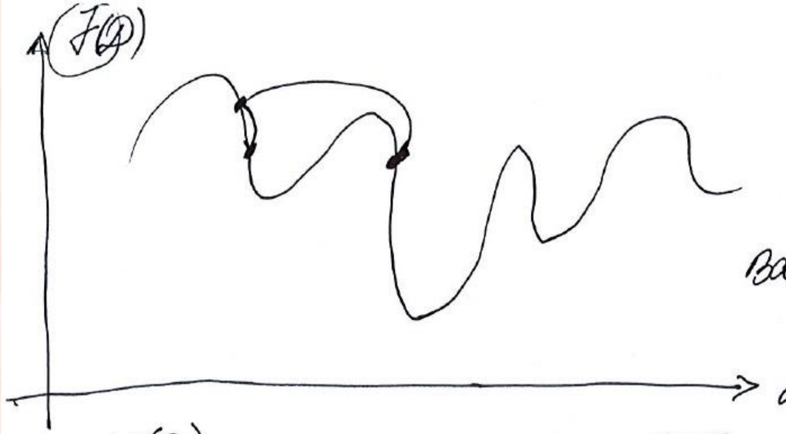
  => the *least-mean-square* (LMS) algorithm.

Batch GD vs. Stochastic GD

- Accuracy:
- Time complexity:
- Memory complexity:
- Online learning:

Batch GD vs. Stochastic GD



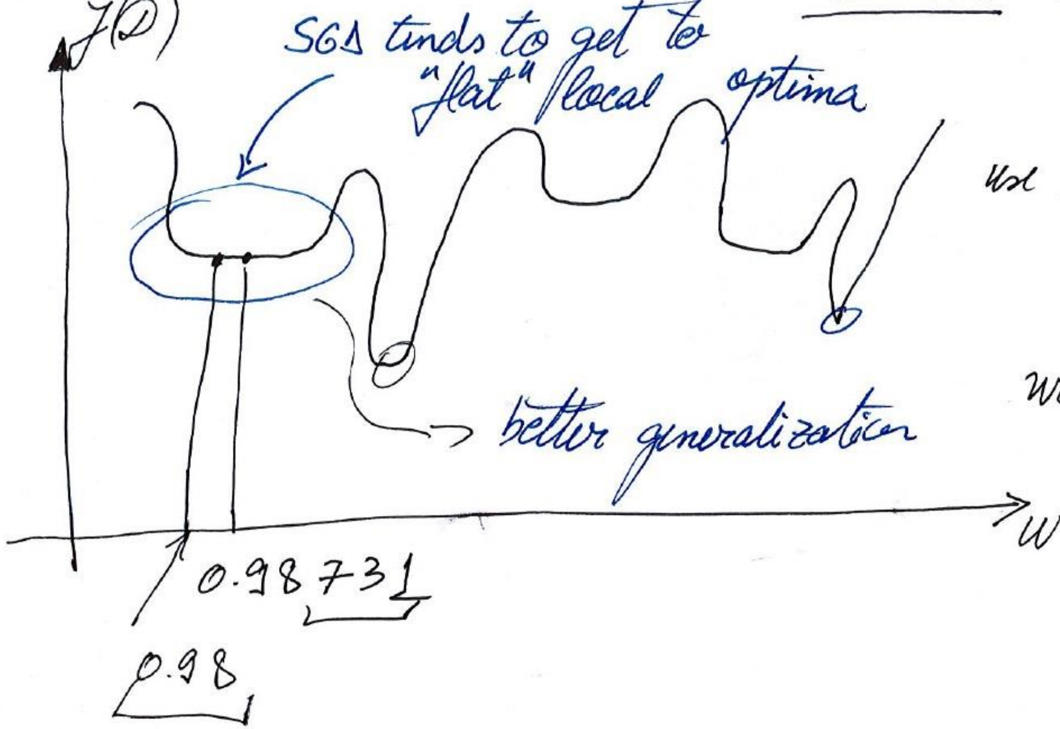


Stochastic w/ mini-batch $\mathcal{B}$ of entire dataset $\mathcal{D}$, $\mathcal{B} \subseteq \mathcal{D}$

Batch GD: Compute $J(\mathcal{D})$ and use $\frac{\partial J(\mathcal{D})}{\partial w}$

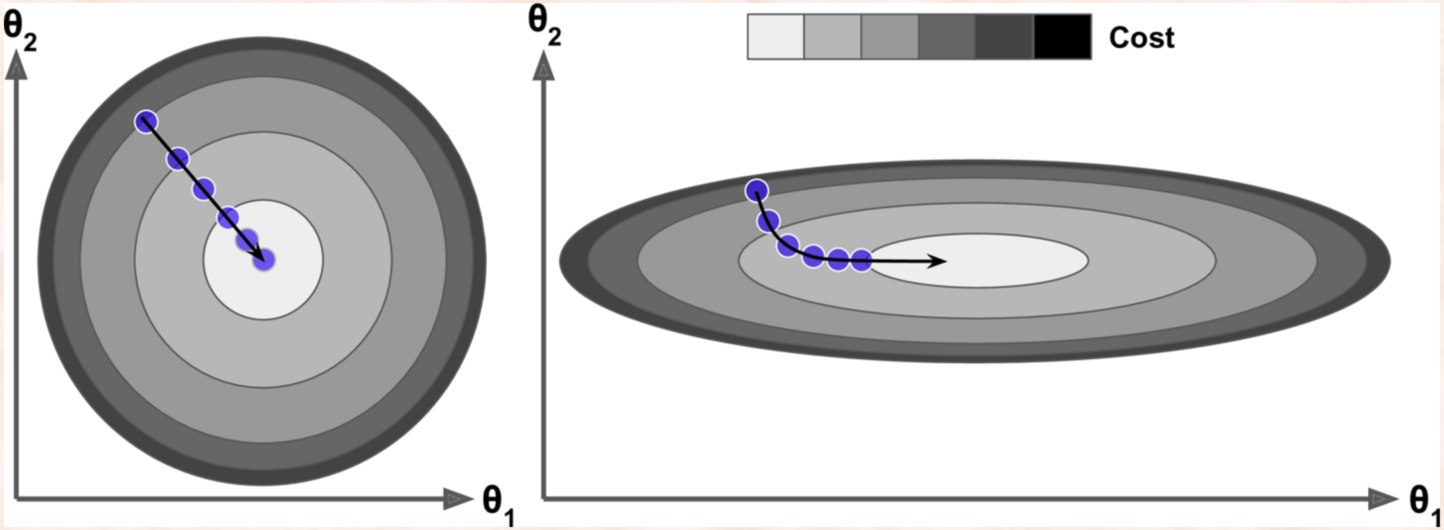
Stochastic: Compute $J(\mathcal{B})$, use $\frac{\partial J(\mathcal{B})}{\partial w}$

use it as a substitute for $\frac{\partial J(\mathcal{D})}{\partial w}$, want to minimize $J(\mathcal{D})$.



Pre-processing Features

- Features may have very different scales, e.g. x_1 = rooms vs. x_2 = size in sq ft.
  - Right (*different scales*): GD goes first towards the bottom of the bowl, then slowly along an almost flat valley.
  - Left (*scaled features*): GD goes straight towards the minimum.



Feature Scaling

- Scaling between [0, 1] or [−1, +1]:
  - For each feature x_j, compute min_j and max_j **over the training examples**.
  - Scale x_j as follows: $\hat{x}_j = \frac{x_j - min_j}{max_j - min_j}$

- Scaling to standard normal distribution:
  - For each feature x_j, compute sample μ_j and sample σ_j **over the training examples**.
  - Scale x_j as follows: $\hat{x}_j = \frac{x_j - \mu_j}{\sigma_j}$

- **Use the same scaling factors at test time**:
  - Clip to min_j and max_j.

n	x_j	$\hat{x}_j$	
1	2	0.2	Train
2	1	0	
3	6	1	
4	8	?	Test

Scaling to [0,1]

$\min_j = 1 \quad \max_j = 6$

$n=1: \quad \hat{x}_j = \dfrac{x_j - \min_j}{\max_j - \min_j} = \dfrac{2-1}{6-1} = \dfrac{1}{5}$

$n=2: \quad \hat{x}_j = \dfrac{1-1}{6-1} = 0$

$n=3: \quad \hat{x}_j = \dfrac{6-1}{6-1} = 1$

$n=4 \quad \hat{x}_j = \dfrac{8-1}{6-1} = \dfrac{7}{5} = 1.4$ ①

clipping: clip to $[0,1]$ ②

$\hat{x}_j = 1$.

Standardization

- sample mean μ_j:

$$\mu_j = \frac{1}{N}\sum_{n=1}^{N} x_j^{(n)}$$

- sample std.dev σ_j:

$$\sigma_j = \sqrt{\frac{1}{N}\sum_{n=1}^{N}\left(x_j^{(n)} - \mu_j\right)^2}$$

- $\hat{x}_j = \dfrac{x_j - \mu_j}{\sigma_j}$

Gradient Descent vs. Normal Equations

- **Gradient Descent**:
  - Need to select learning rate η.
  - May need many iterations:
    - Can do *Early Stopping* on validation data for regularization.
  - Scalable when number of training examples N is large.

- **Normal Equations**:
  - No iterations => easy to code.
  - Computing $(X^TX)^{-1}$ has cubic time complexity => slow for large N.
  - X^TX may be singular:
    1. Redundant (linearly dependent) features.
    2. #features > #examples => do *feature selection* or *regularization*.

Implementation: Vectorization

- **Version 1**: Compute gradient component-wise.

$$\nabla J(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \left(\mathbf{w}^T\mathbf{x}^{(n)} - y_n\right)\mathbf{x}^{(n)}$$

$$\hat{y}_n = \mathbf{w}^T\mathbf{x}^{(n)}$$

```
grad = np.zeros(K)
for n in range(N):
    h = w.dot(X[n])  // This NumPy code assumes examples stored in rows of X.
    temp = h − y[n]
    for k in range(K):
      grad[k] = grad[k]+ temp * X[n,k]
for k in range(K):
  grad[k] = grad[k] / N
```

Implementation: Vectorization

- **Version 2**: Compute gradient, partially vectorized.

$$\hat{y}_n = \mathbf{w}^T \mathbf{x}^{(n)}$$

$$\nabla J(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} \left(\mathbf{w}^T \mathbf{x}^{(n)} - y_n \right) \mathbf{x}^{(n)}$$

```
grad = np.zeros(K)
for n in range(N):   // This NumPy code assumes examples stored in rows of X.
    grad = grad + (w.dot(X[n])) − y[n]) * X[n]
grad = grad / N
```

Implementation: Vectorization

- **Version 3**: Compute gradient, vectorized.

$$\hat{y}_n = \mathbf{w}^T \mathbf{x}^{(n)}$$

$$\nabla J(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} \left(\mathbf{w}^T \mathbf{x}^{(n)} - y_n \right) \mathbf{x}^{(n)}$$

grad = X.T.dot(X.dot(**w**) − **y**) / N

NumPy code above assumes examples stored in columns of X

Implementation: Gradient Checking

- Want to minimize $J(\theta)$, where θ is a scalar.

- Mathematical definition of derivative:

$$\frac{d}{d\theta}J(\theta) = \lim_{\varepsilon \to \infty} \frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon}$$

- Numerical approximation of derivative:

$$\frac{d}{d\theta}J(\theta) \approx \frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon} \qquad \text{where } \varepsilon = 0.0001$$

Implementation: Gradient Checking

- If $\boldsymbol{\theta}$ is a vector of parameters $\boldsymbol{\theta}_i$,
  - Compute numerical derivative with respect to each $\boldsymbol{\theta}_i$.
  - Aggregate all derivatives into numerical gradient $G_{\text{num}}(\boldsymbol{\theta})$.

- Compare numerical gradient $G_{\text{num}}(\boldsymbol{\theta})$ with implementation of gradient $G_{\text{imp}}(\boldsymbol{\theta})$:

$$\frac{\left\| G_{num}(\theta) - G_{imp}(\theta) \right\|}{\left\| G_{num}(\theta) + G_{imp}(\theta) \right\|} \leq 10^{-6}$$

Supplemental Topics

Gradient Descent Optimization Algorithms

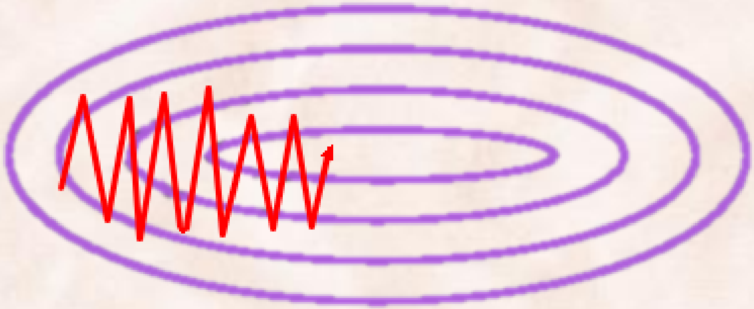
- **Momentum**.
- **Nesterov Accelerated Gradient** (NAG).
- Adaptive learning rates methods:
  - Idea is to perform larger updates for infrequent params and smaller updates for frequent params, by accumulating previous gradient values for each parameter.
    - **Adagrad**:
      - Divide update by sqrt of sum of squares of past gradients.
    - **Adadelta**.
    - **RMSProp**.
    - **Adaptive Moment Estimation** (Adam)

Gradient Descent & Momentum

Vanilla Gradient Descent:

$$\mathbf{v}^{\tau+1} = \eta \nabla J(\mathbf{w}^{\tau})$$

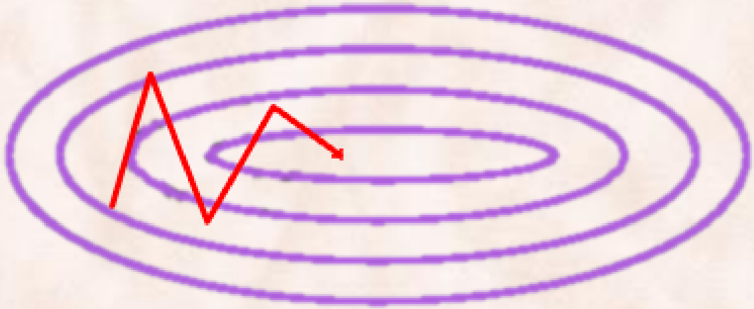
$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \mathbf{v}^{\tau+1}$$



Gradient Descent w/ Momentum:

$$\mathbf{v}^{\tau+1} = \gamma \mathbf{v}^{\tau} + \eta \nabla J(\mathbf{w}^{\tau})$$

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \mathbf{v}^{\tau+1}$$



γ is usually set to 0.9 or similar.

The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions.

Momentum & Nesterov Accelerated Gradient

GD with Momentum:

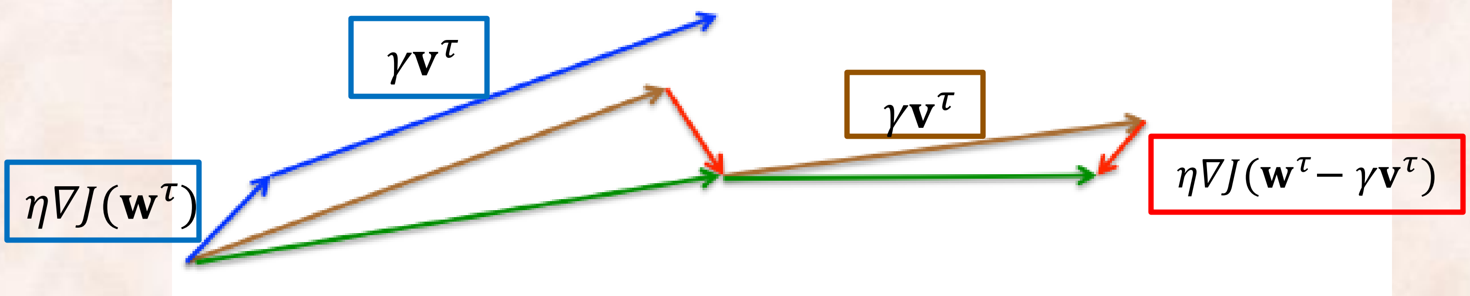
$$\mathbf{v}^{\tau+1} = \gamma \mathbf{v}^{\tau} + \eta \nabla J(\mathbf{w}^{\tau})$$

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \mathbf{v}^{\tau+1}$$

Nesterov Accelerated Gradient:

$$\mathbf{v}^{\tau+1} = \gamma \mathbf{v}^{\tau} + \eta \nabla J(\mathbf{w}^{\tau} - \gamma \mathbf{v}^{\tau})$$

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \mathbf{v}^{\tau+1}$$



Nesterov update (Source: G. Hinton's lecture 6c)

By making an anticipatory update, NAGs prevents GD from going too fast => significant improvements when training RNNs.

AdaGrad

- Optimized for problems with sparse features.
- Per-parameter learning rate: make smaller updates for params that are updated more frequently:

$$w_i = w_i - \eta \frac{g_{t,i}}{\sqrt{\epsilon + G_{t,i}}} \quad \text{where } G_{t,i} = \sum_{\tau=1}^{t} g_{\tau,i}^2$$

$$g_{t,i} = \frac{\partial J(\mathbf{w})}{\partial w_i}$$

- Require less tuning of the learning rate compared with SGD.

RMSProp

- Element-wise gradient: $g_i^t = \nabla_{w_i} J(\mathbf{w}_t)$
- Gradient is $\mathbf{g}_t = [g_1^t, g_2^t, \ldots, g_K^t]$
- Element-wise square gradient: $\mathbf{g}_t^2 = \mathbf{g}_t \circ \mathbf{g}_t$

RMSProp:

$$\mathrm{E}_t[\mathbf{g}^2] = \gamma \mathrm{E}_{t-1}[\mathbf{g}^2] + (1-\gamma)\,\mathbf{g}_t^2$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{\sqrt{\mathrm{E}_t[\mathbf{g}^2] + \epsilon}}\mathbf{g}_t$$

γ *is usually set to* 0.9, η is set to 0.001

Adam: Adaptive Moment Estimation

- Maintain an exponentially decaying average of past gradients (1st m.) and past squared gradients (2nd m.):
  1) $\mathbf{m}_t = \beta_1 \, \mathbf{m}_{t-1} + (1 - \beta_1) \, \mathbf{g}_t$
  2) $\mathbf{v}_t = \beta_1 \, \mathbf{v}_{t-1} + (1 - \beta_1) \, \mathbf{g}_t^2$

- Biased towards 0 during initial steps, use bias-corrected first and second order estimates:
  1) $\widehat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t}$
  2) $\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t}$

Adam: Adaptive Moment Estimation

- First and second moment:

$$\mathbf{m}_t = \beta_1 \, \mathbf{m}_{t-1} + (1 - \beta_1) \, \mathbf{g}_t$$

$$\mathbf{v}_t = \beta_1 \, \mathbf{v}_{t-1} + (1 - \beta_1) \, \mathbf{g}_t^2$$

- Bias-correction:

$$\widehat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t} \text{ and } \hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t}$$

Adam:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_t + \epsilon}} \widehat{\mathbf{m}}_t$$

Visualization

- Adagrad, RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances.
  - Insofar, **Adam** might be the best overall choice.