CS 6890: Deep Learning

Gradient Descent Algorithms

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

Machine Learning is Optimization

- Parametric ML involves minimizing an **objective function** $J(\mathbf{w})$:
  - Also called **cost function**, **loss function**, or **error 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**:

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

  - $\mathbf{g}$ 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**:

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

- 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 $\mathbf{g} = -\nabla J(\mathbf{w})$ => the **gradient descent** update:

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

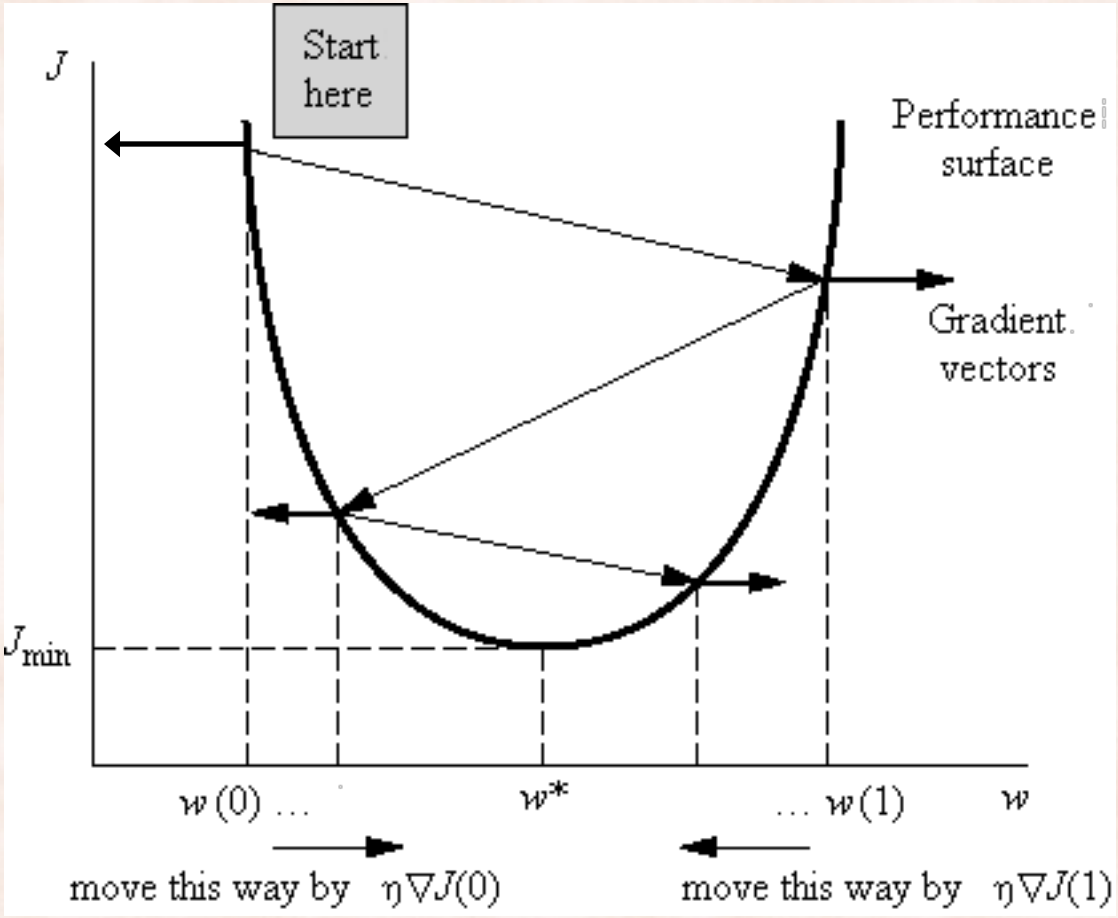
Gradient Descent Algorithm

- 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*:

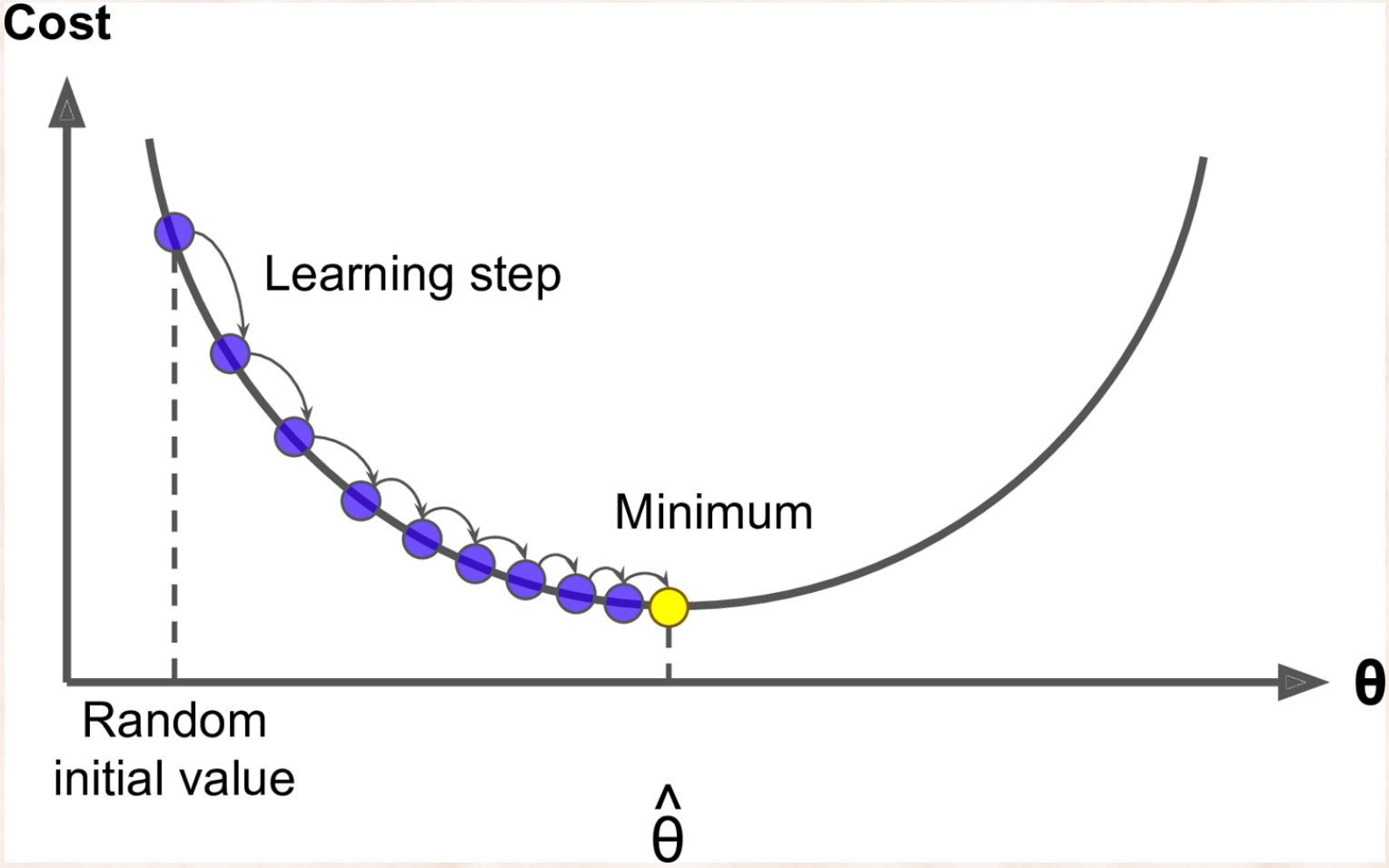
$$\nabla J(\mathbf{w}) = \left[\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \ldots, \frac{\partial J}{\partial w_n}\right]$$

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

Gradient Descent: Large Updates



Gradient Descent: Small Updates



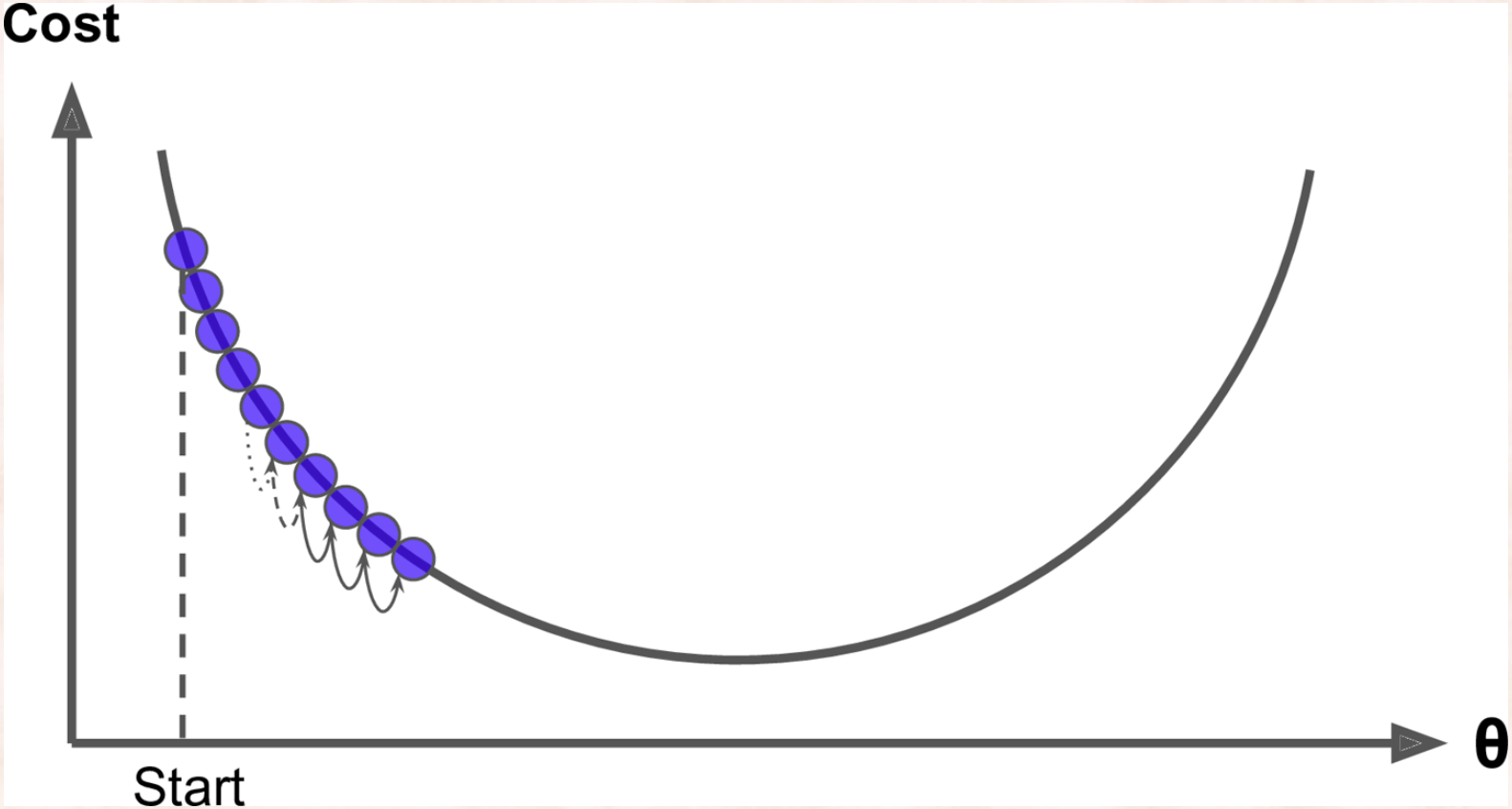
https://www.safaribooksonline.com/library/view/hands-on-machine-learning

The Learning Rate

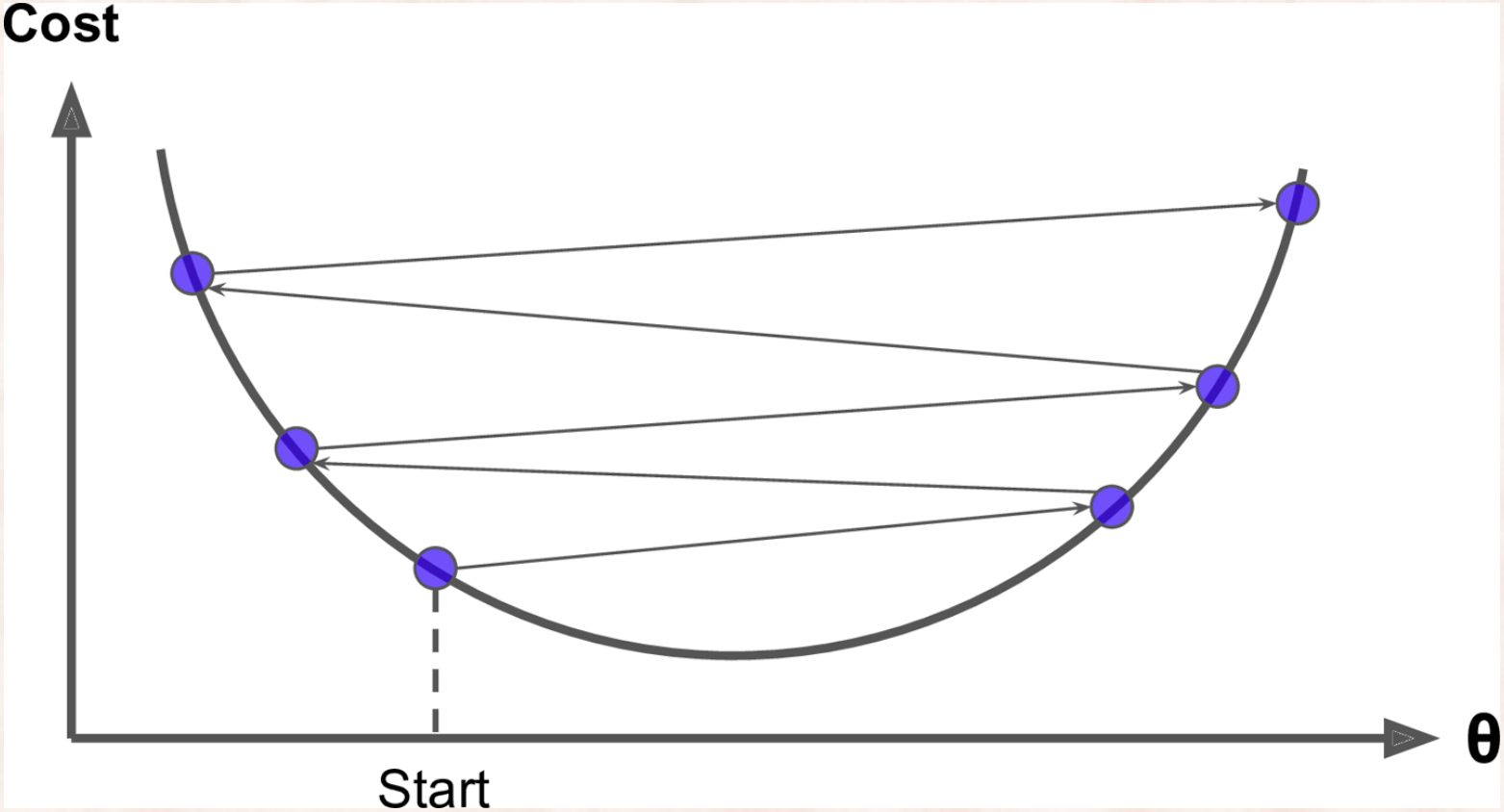
1. Set **learning rate** $\eta = 0.001$ (or other small value).
2. Start with some guess for $\mathbf{w}^0$, set $\tau = 0$.
3. Repeat for epochs E or until J does not improve:
4. $\quad \tau = \tau + 1.$
5. $\quad \mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta \nabla J(\mathbf{w}^{\tau})$

- 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.

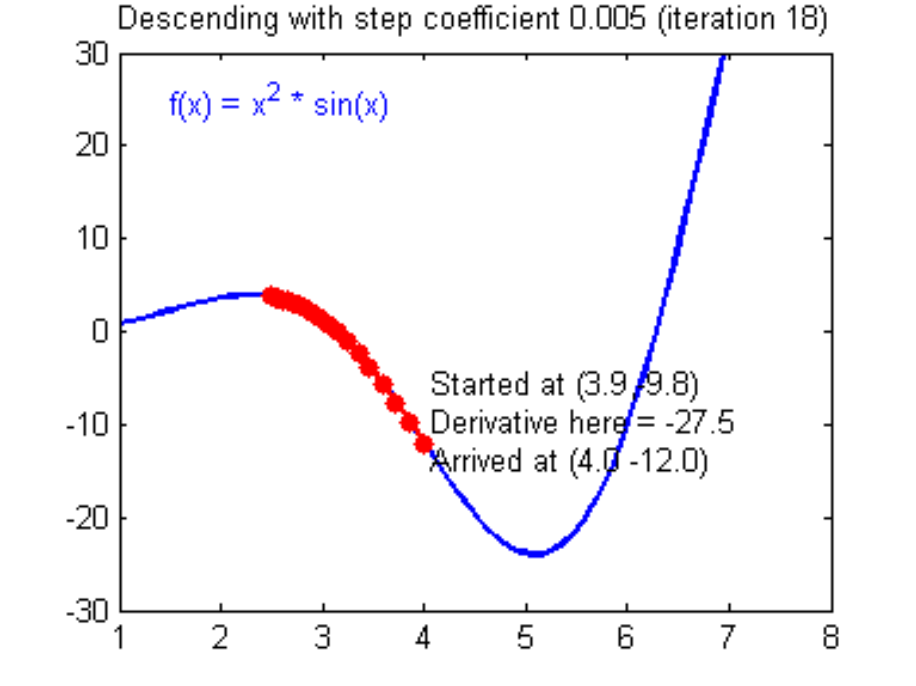
Learning Rate too Small

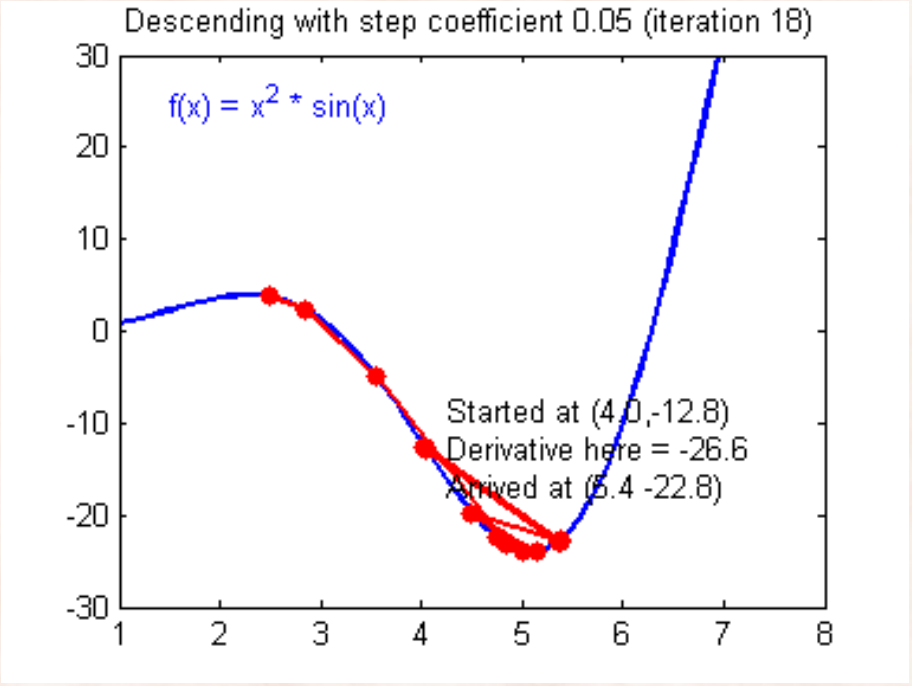


Learning Rate too Large



Learning Rates vs. GD Behavior





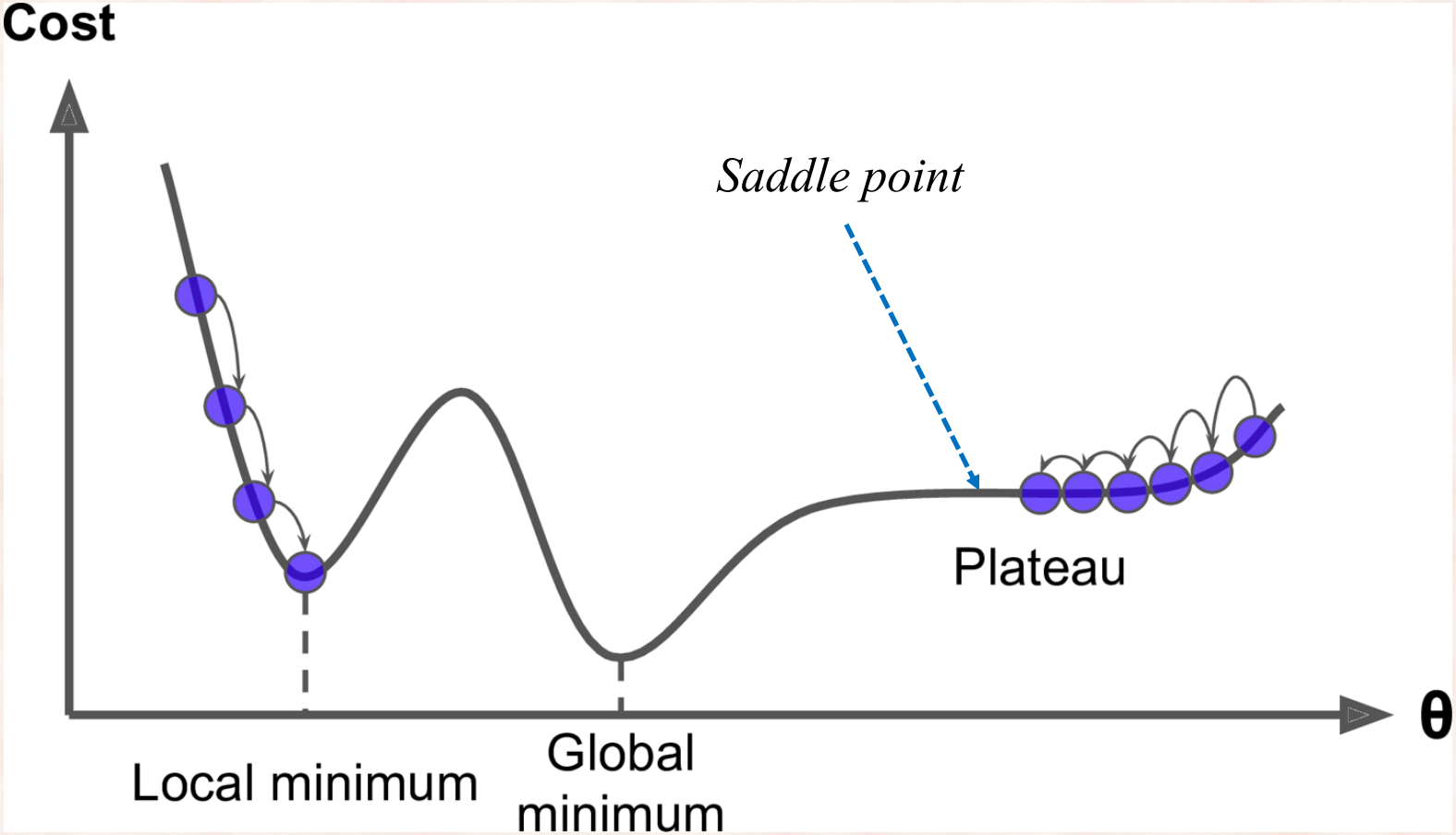
http://scs.ryerson.ca/~aharley/neural-networks/

The Learning Rate

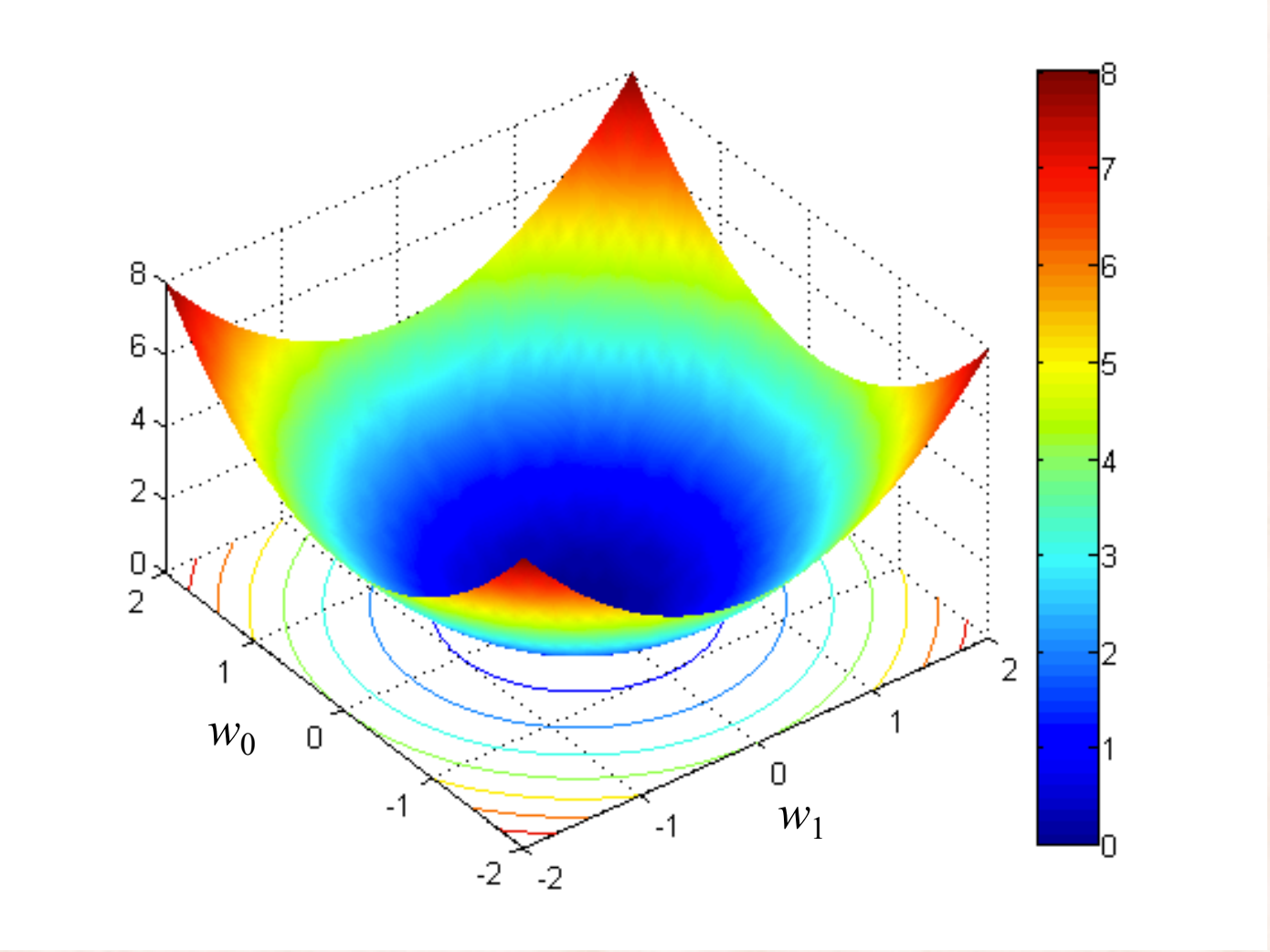
- How big should the **learning rate** be?
  - If learning rate too big => oscillating behavior.
  - If learning rate too small => hinders convergence.

- 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.

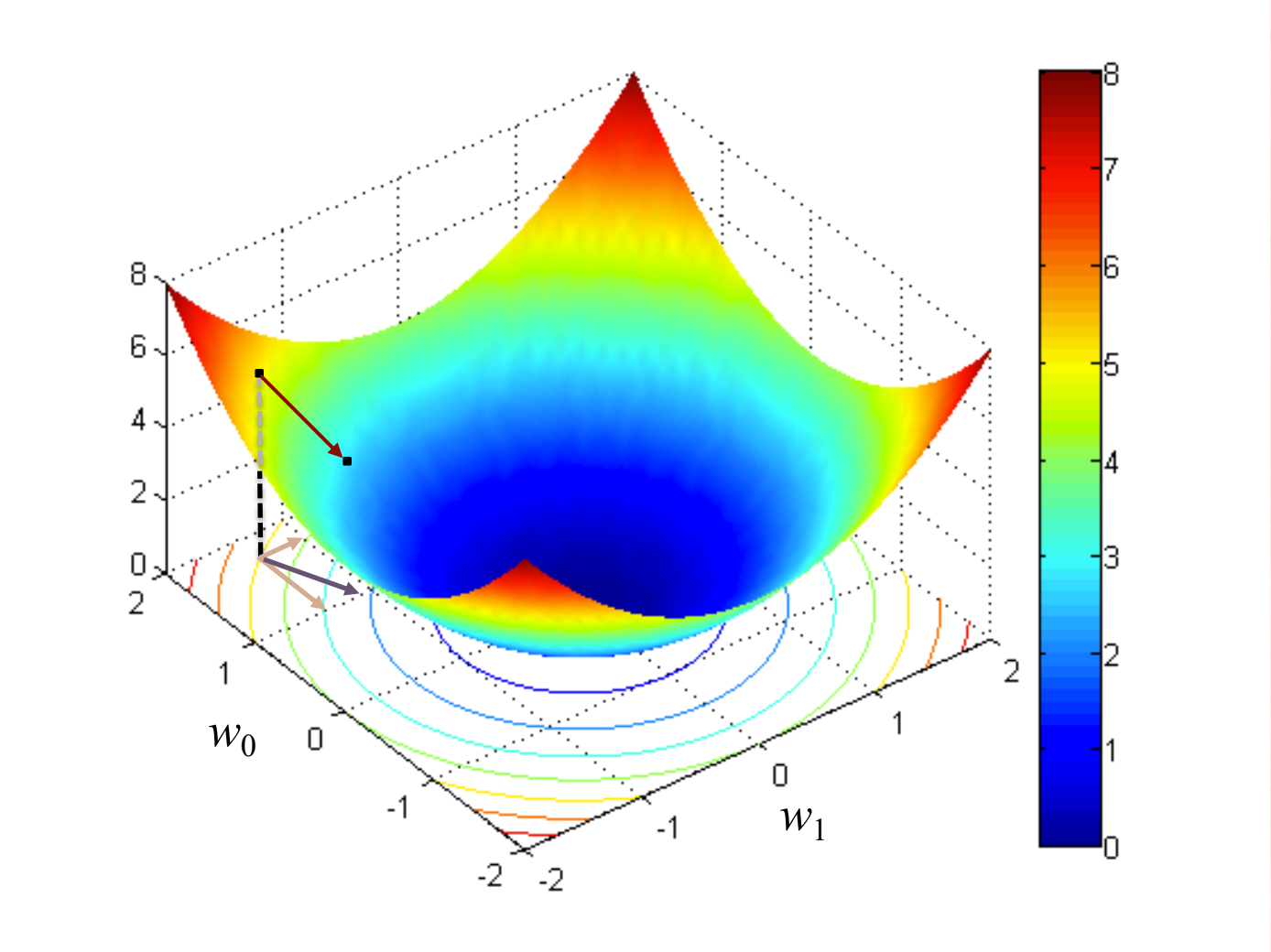
Gradient Descent: Nonconvex Objective



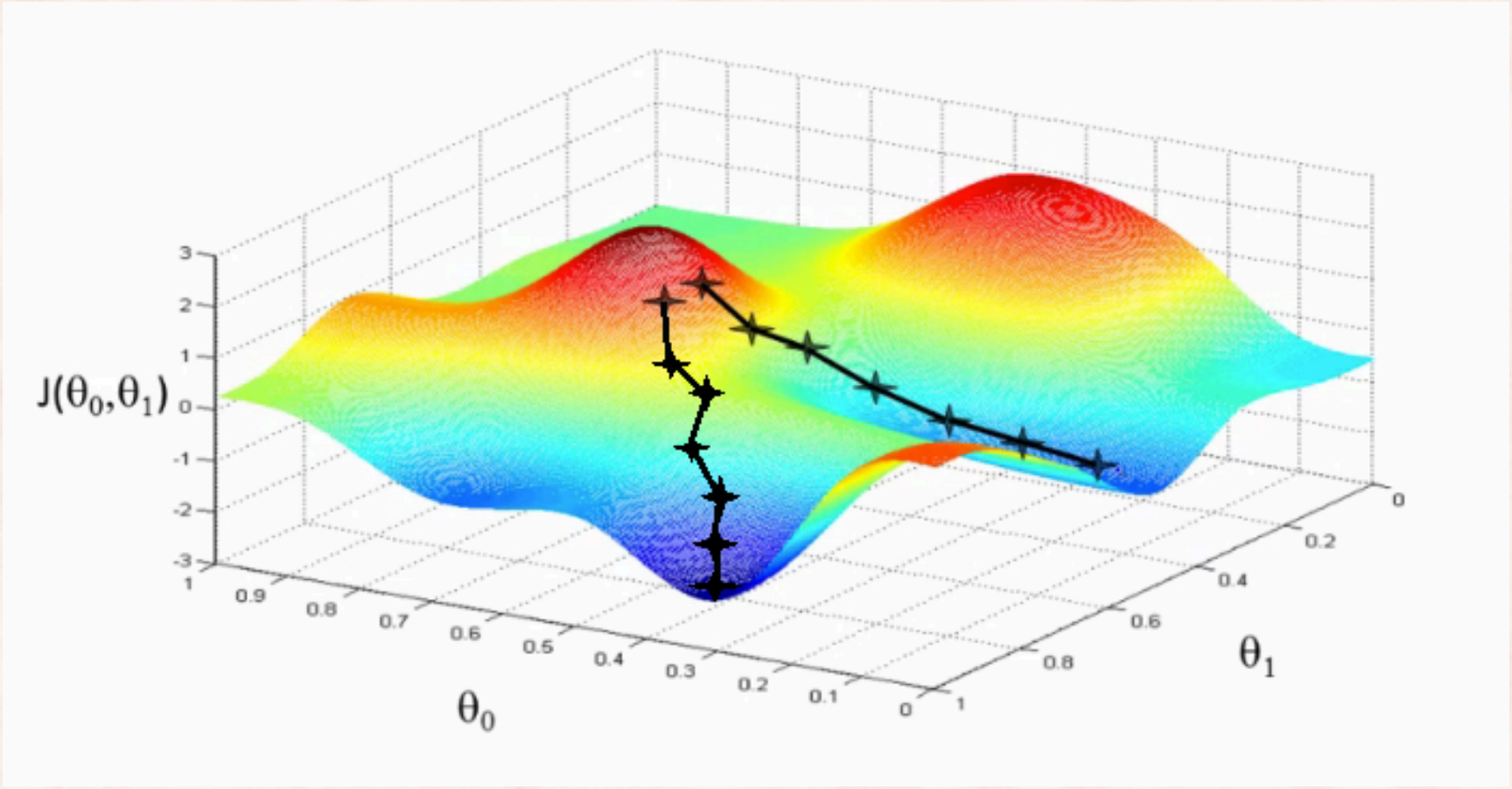
Convex Multivariate Objective



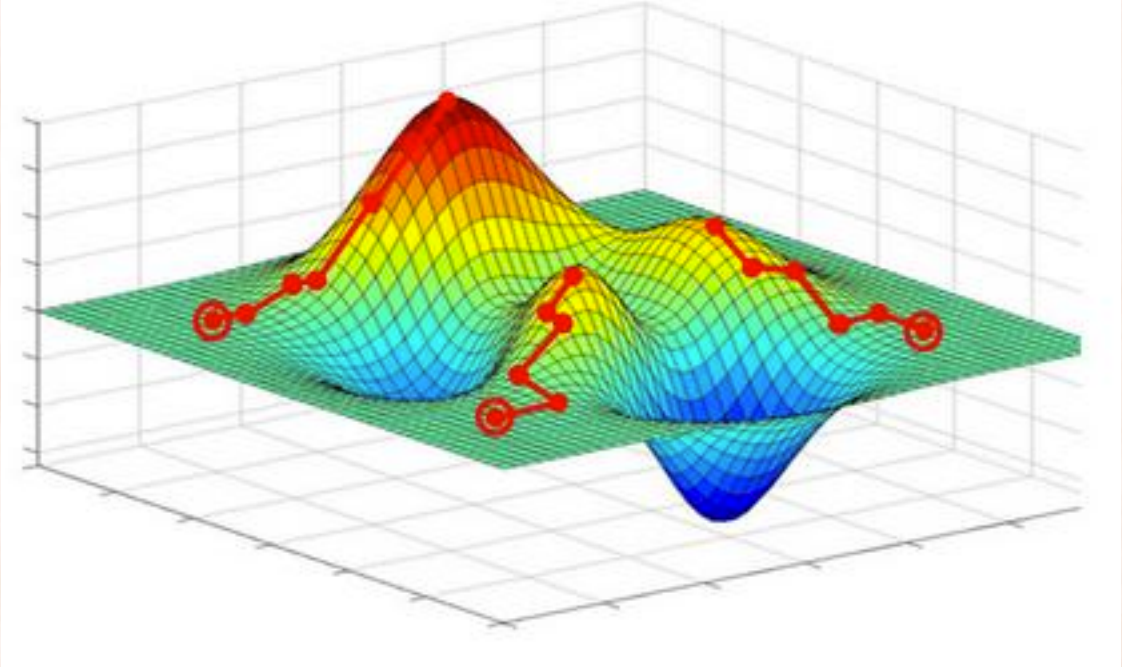
Gradient Step and Contour Lines



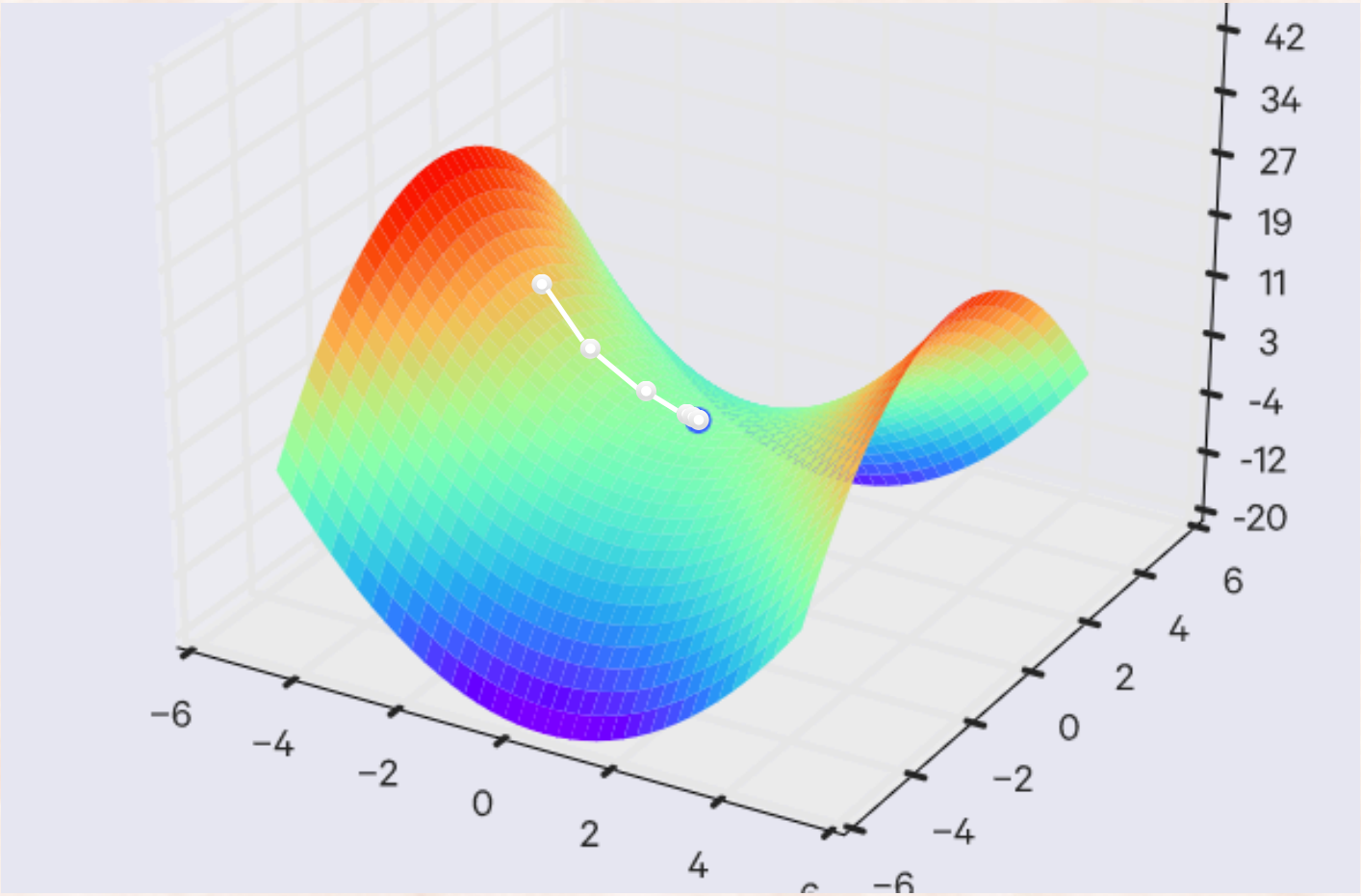
Gradient Descent: Nonconvex Objectives



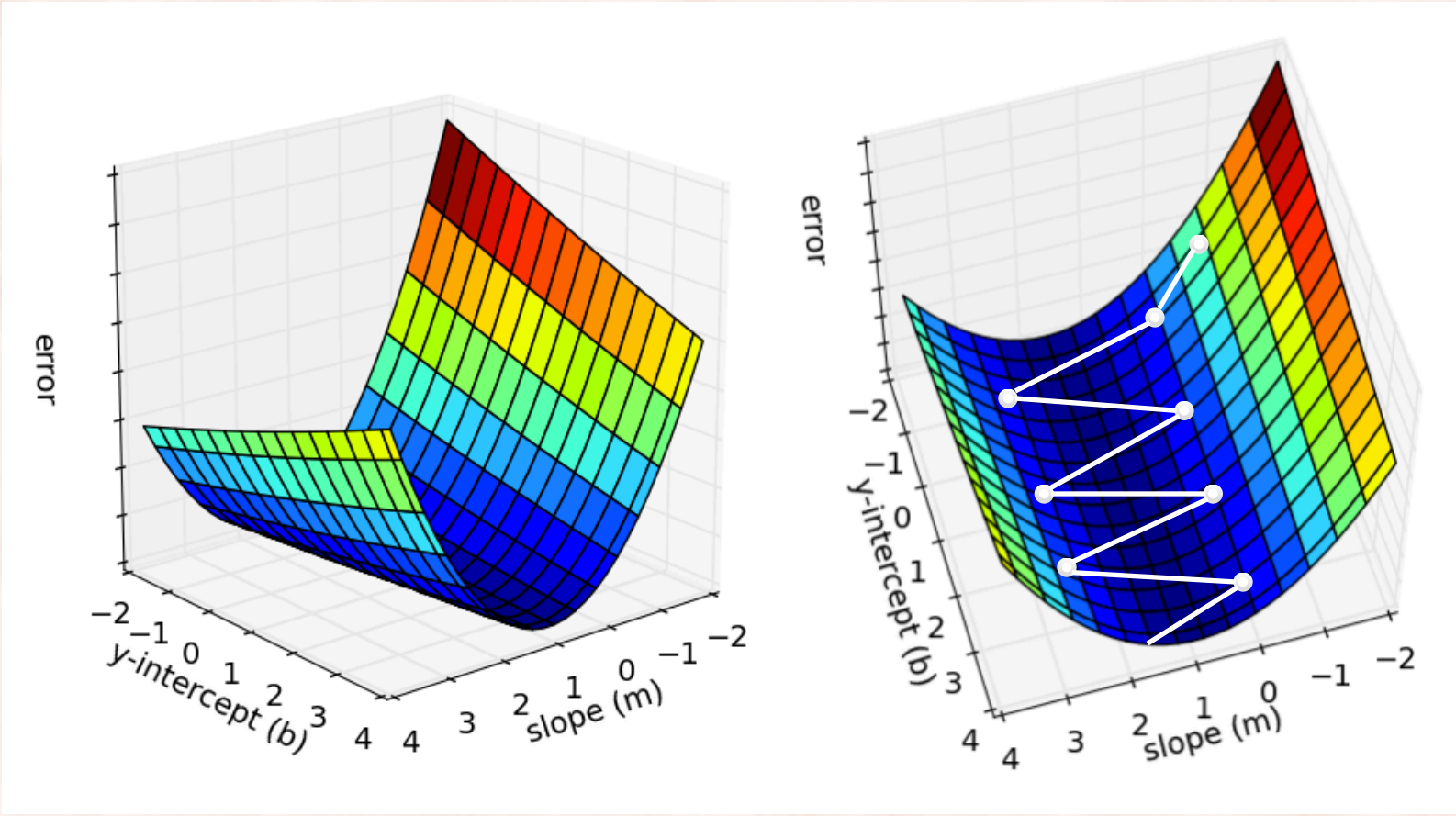
Gradient Descent & Plateaus



Gradient Descent & Saddle Points



Gradient Descent & Ravines



Gradient Descent & Ravines

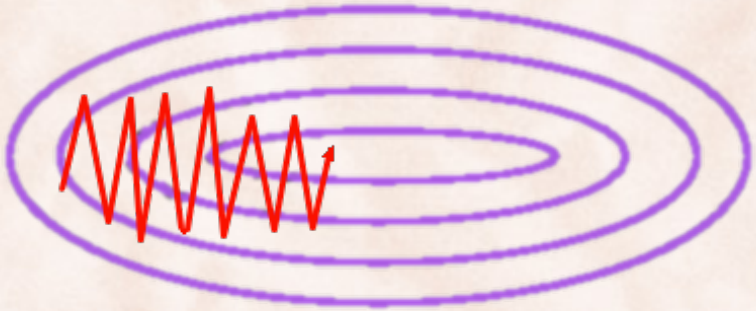
- **Ravines** are areas where the surface curves much more steeply in one dimension than another.
  - Common around local optima.
  - GD oscillates across the slopes of the ravines, making slow progress towards the local optimum along the bottom.
- Use **momentum** to help accelerate GD in the relevant 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 reduces the risk of getting stuck in local minima.

Gradient Descent & Momentum

Vanilla Gradient Descent:

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

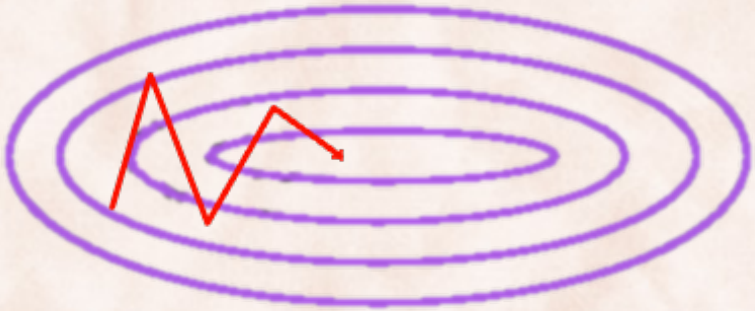
$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \mathbf{v}^{\tau+1}$$



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.*

Momentum & Nesterov Accelerated Gradient

GD with Momentum:

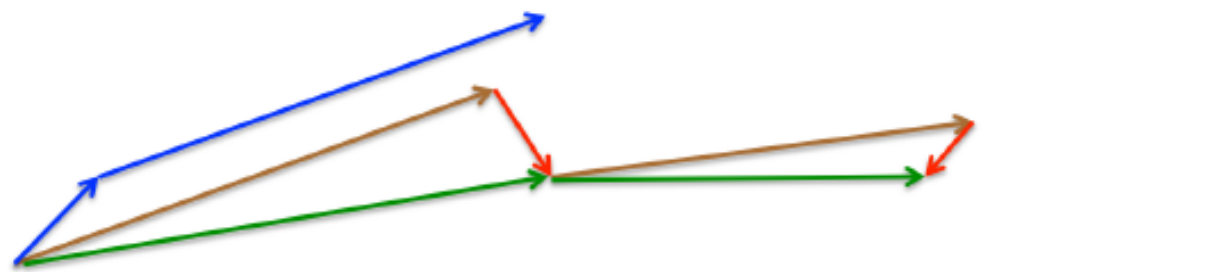
$$\mathbf{v}^{\tau+1} = \gamma \mathbf{v}^{\tau} + \eta \nabla J(\mathbf{w}^{\tau})$$

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

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.

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)

AdaGrad

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

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

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

- 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) $\mathbf{v}_t = \beta_2 \, \mathbf{v}_{t-1} + (1 - \beta_2) \, \mathbf{g}_t^2$

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

Adam: Adaptive Moment Estimation

- First and second moment:

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

$$\mathbf{v}_t = \beta_2 \, \mathbf{v}_{t-1} + (1 - \beta_2) \, \mathbf{g}_t^2$$

- Bias-correction:

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

Adam:

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

Adam : Adaptive Moment Estimation

- Authors (Kingma & Ba) proposed default values:
  - $\beta_1 = 0.9$
  - $\beta_2 = 0.999$
  - $\eta = 10^{-8}$

- However, Dozat & Manning in "Deep Biaffine Attention for Neural Dependency Parsing" (ICLR 2017):
  - We find that the value for β_2 recommended by Kingma & Ba – which controls the decay rate for this moving average – is too high for this task (and we suspect more generally). When this value is very large, the magnitude of the current update is heavily influenced by the larger magnitude of gradients very far in the past, with the effect that the optimizer can't adapt quickly to recent changes in the model. Thus we find that setting β_2 to .9 instead of .999 makes a large positive impact on final performance."

Adaptive methods vs. SGD

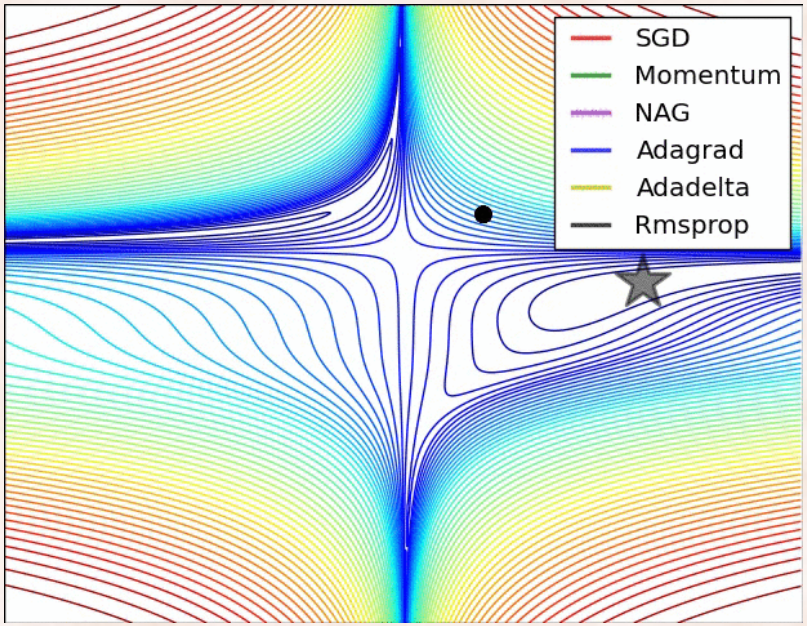
- However, Wilson et al. in "The Marginal Value of Adaptive Gradient Methods in Machine Learning" (NIPS 2017) find that:
  - Solutions found by adaptive methods generalize worse (often *significantly* worse) than SGD, even when these solutions have better training performance.
    - These results suggest that practitioners should reconsider the use of adaptive methods to train neural networks.

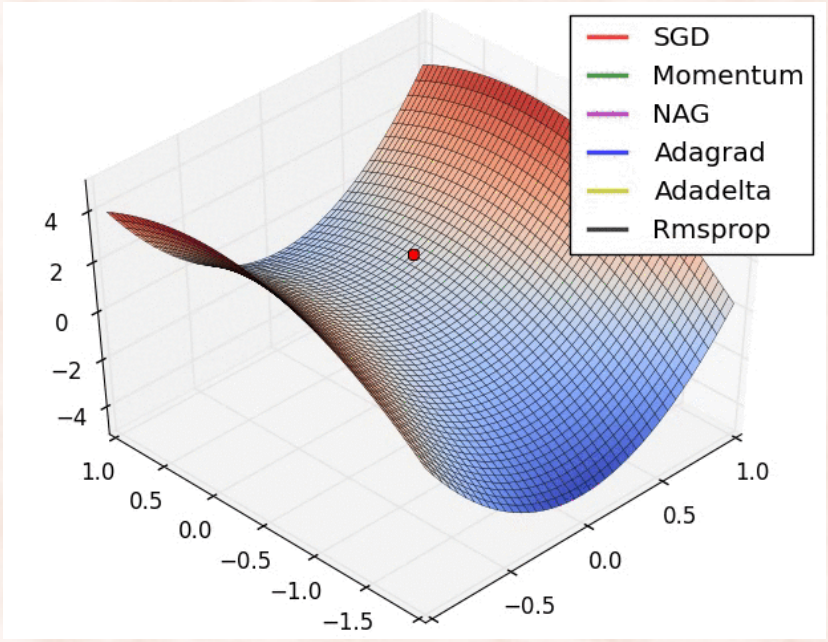
Adaptive methods vs. SGD

- Luo et al. in "Adaptive Gradient Methods with Dynamic Bound of Learning Rate" (ICLR 2019) propose **AdaBound**:
  - We provide new variants of Adam and AMSGrad, called AdaBound and AMSBound respectively, which employ dynamic bounds on learning rates to achieve a gradual and smooth transition from adaptive methods to SGD and give a theoretical proof of convergence.
  - Experimental results show that new variants can eliminate the generalization gap between adaptive methods and SGD and maintain higher learning speed early in training at the same time. Moreover, they can bring significant improvement over their prototypes, especially on complex deep networks.

Visualization

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





Variants of Gradient Descent

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

- 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

- Sum-of-squares error:

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

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

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

Stochastic Gradient Descent

- Sum-of-squares error:

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

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

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

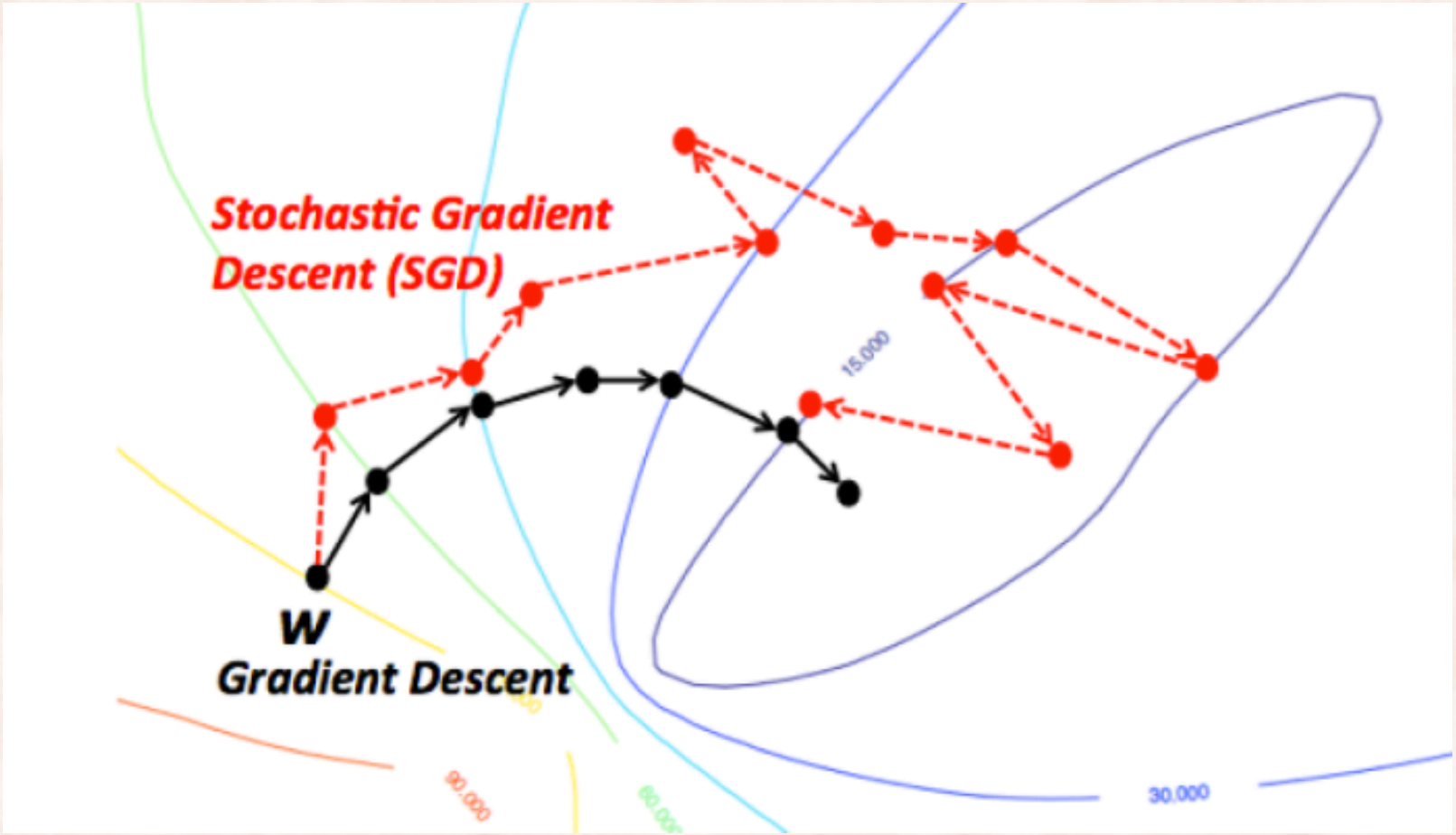
- Update parameters **w** after each example, sequentially:

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

Batch GD vs. Stochastic GD

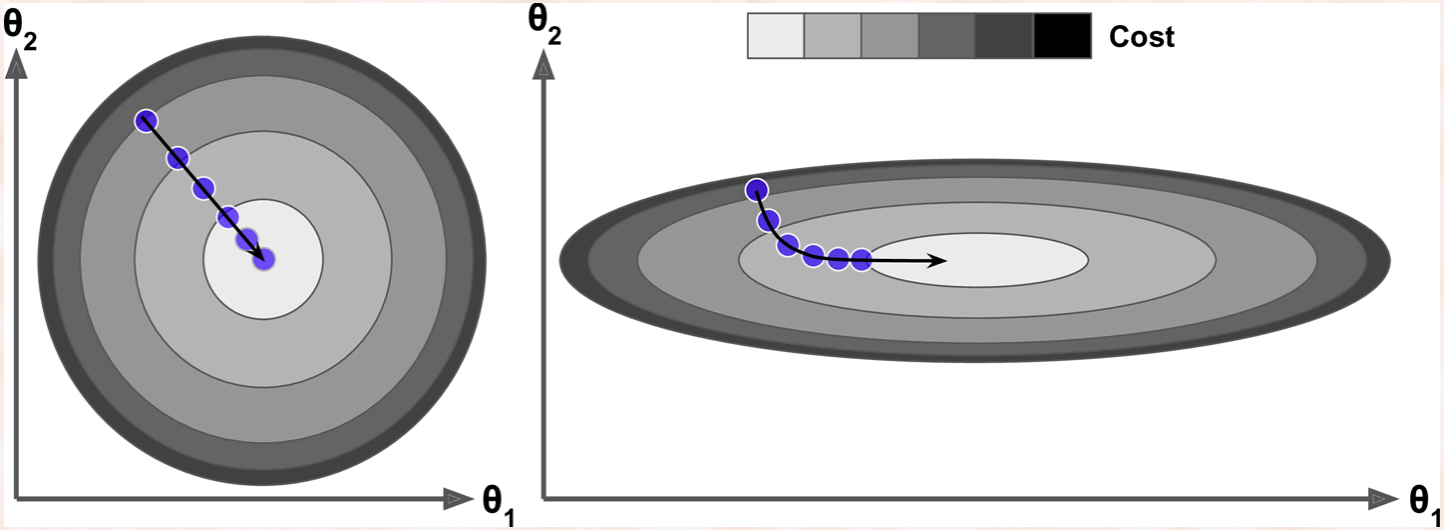
- Accuracy:
- Time complexity:
- Memory complexity:
- Online learning:

Batch GD vs. Stochastic GD



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 $x^{(n)}_j$ as follows:

- Scaling to standard normal distribution:
  - For each feature x_j, compute sample μ_j and sample σ_j over the training examples.
  - Scale $x^{(n)}_j$ as follows:

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:

$$\frac{d}{d\theta}J(\theta) \approx \frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon}$$ 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 $G_{\text{imp}}(\boldsymbol{\theta})$:

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