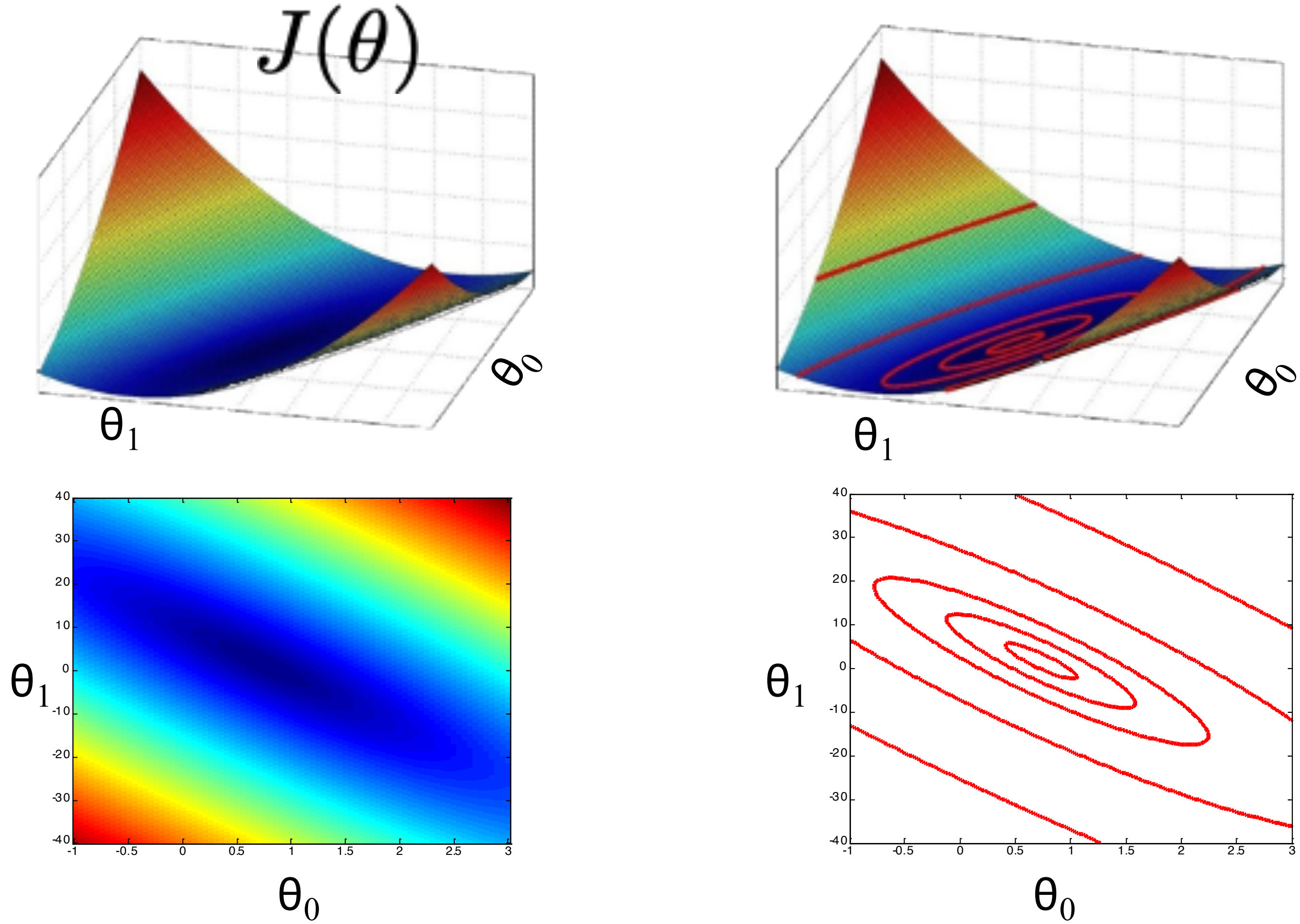
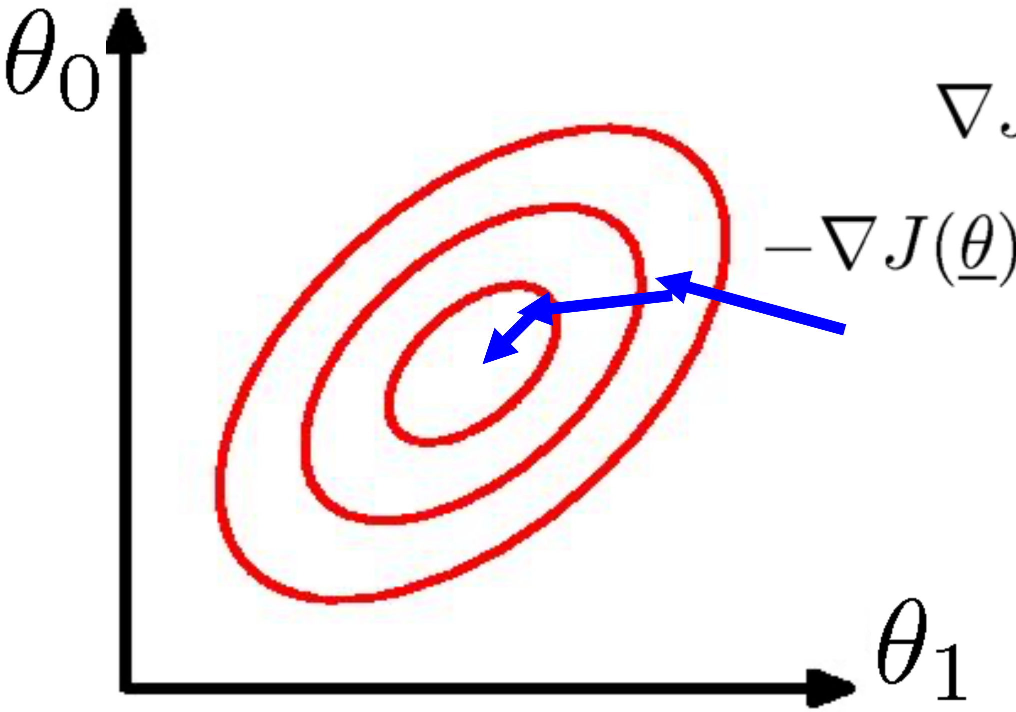
Reminder: the cost function



Reminder: Gradient descent

- Gradient vector

$$\nabla J(\underline{\theta}) = \left[\frac{\partial J(\underline{\theta})}{\partial \theta_0} \quad \frac{\partial J(\underline{\theta})}{\partial \theta_1} \quad \ldots \right]$$



Indicates direction of steepest ascent
(negative = steepest descent)

Gradient descent

Initialization

Step size α

- Can change as a function of iteration

Gradient direction

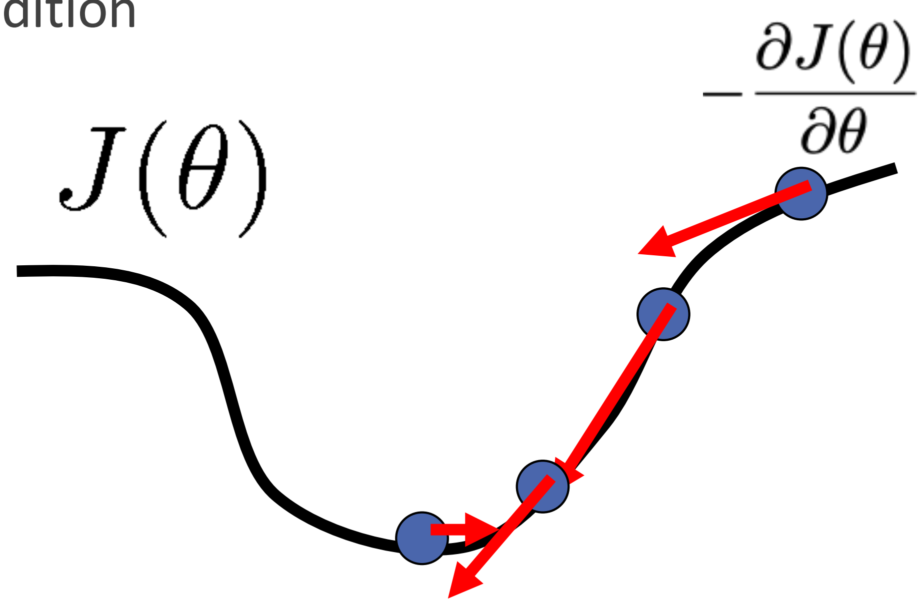
Stopping condition

```
Initialize θ

Do{

 θ ← θ - α∇θJ(θ)

} while (α||∇θJ|| > ε)
```



Gradient for the MSE

MSE $$J(\underline{\theta}) = \frac{1}{m} \sum_j (y^{(j)} - \underline{\theta} \cdot \underline{x}^{(j)T})^2$$

∇J = ?
$$J(\underline{\theta}) = \frac{1}{m} \sum_j (\underbrace{y^{(j)} - \theta_0 \underline{x}_0^{(j)} - \theta_1 \underline{x}_1^{(j)} - \dots}_{e_j(\theta)})^2$$

$$\frac{\partial J}{\partial \theta_0} = \frac{\partial}{\partial \theta_0} \frac{1}{m} \sum_j (\ e_j(\theta)\ )^2$$

$$= \frac{1}{m} \sum_j \ \frac{\partial}{\partial \theta_0} (\ e_j(\theta)\ )^2$$

$$= \frac{1}{m} \sum_j \ 2 e_j(\theta) \ \frac{\partial}{\partial \theta_0} e_j(\theta)$$

$$\frac{\partial}{\partial \theta_0} e_j(\theta) = \underbrace{\cancel{\frac{\partial}{\partial \theta_0} y^{(j)}}}_{0} - \frac{\partial}{\partial \theta_0} \theta_0 x_0^{(j)} - \underbrace{\cancel{\frac{\partial}{\partial \theta_0} \theta_1 x_1^{(j)}}}_{0} - \dots$$

$$= -x_0^{(j)}$$

Gradient for the MSE

MSE $$J(\underline{\theta}) = \frac{1}{m}\sum_j (y^{(j)} - \underline{\theta} \cdot \underline{x}^{(j)T})^2$$

∇J = ? $$J(\underline{\theta}) = \frac{1}{m}\sum_j (y^{(j)} \underbrace{- \theta_0 \underline{x}_0^{(j)} - \theta_1 \underline{x}_1^{(j)} - \ldots}_{e_j(\theta)})^2$$

$$\nabla J(\theta) = [\frac{\partial J}{\partial \theta_0}, \frac{\partial J}{\partial \theta_1}, \ldots]$$

$$= [\frac{-2}{m}\sum_j e_j(\theta) x_0^{(j)}, \frac{-2}{m}\sum_j e_j(\theta) x_1^{(j)}, \ldots]$$

$$= \frac{-2}{m}\sum_j e_j(\theta)\, [x_0^{(j)}, x_2^{(j)}, \ldots, x_n^{(j)}] .$$

Gradient for the MSE

Rewrite using matrix form

$$\nabla J(\underline{\theta}) = -\frac{2}{m} \sum_j \underbrace{(y^{(j)} - \underline{\theta} \cdot \underline{x}^{(j)T})}_{\text{Error magnitude \& direction for datum j}} \cdot \underbrace{[x_0^{(j)} x_1^{(j)} \ldots]}_{\text{Sensitivity to each } \theta_i}$$

$$\underline{\theta} = [\theta_0, \ldots, \theta_n]$$

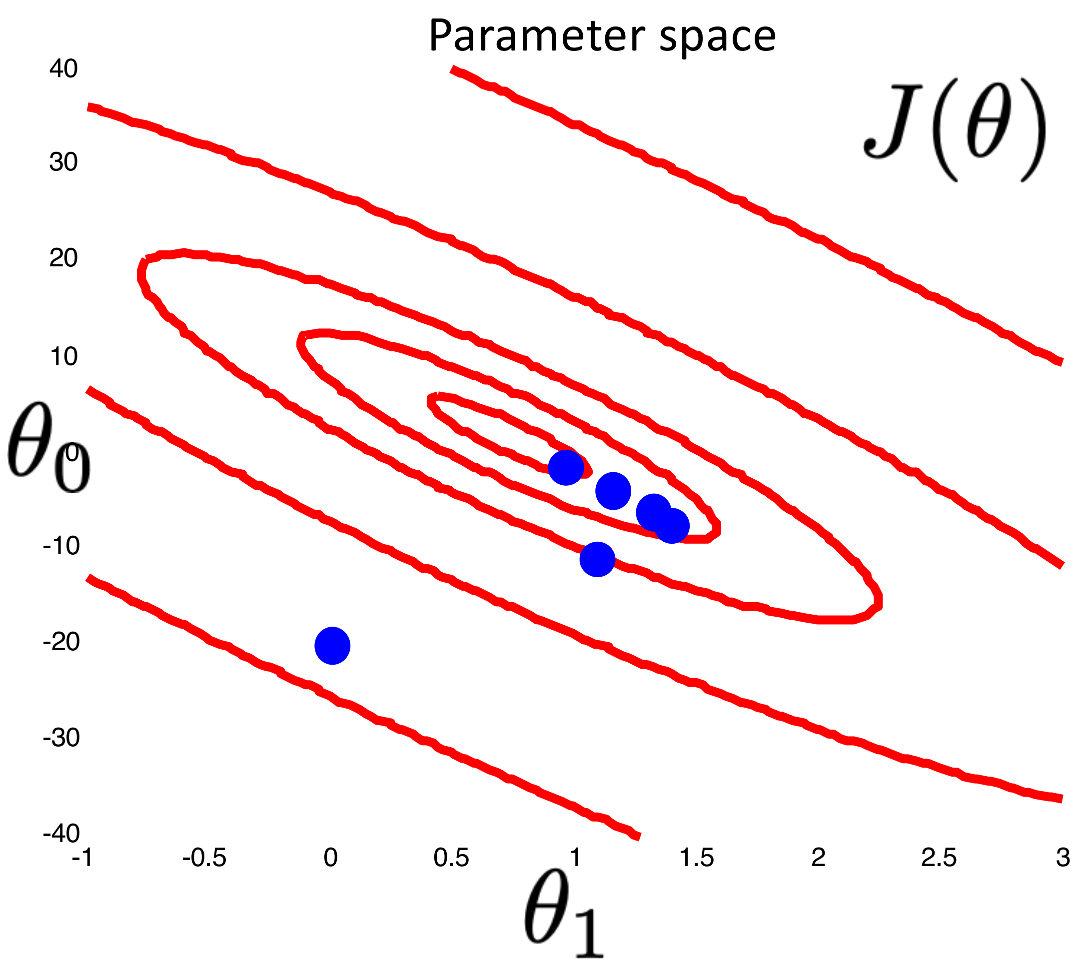
$$\underline{y} = \left[y^{(1)} \ldots, y^{(m)} \right]^T$$

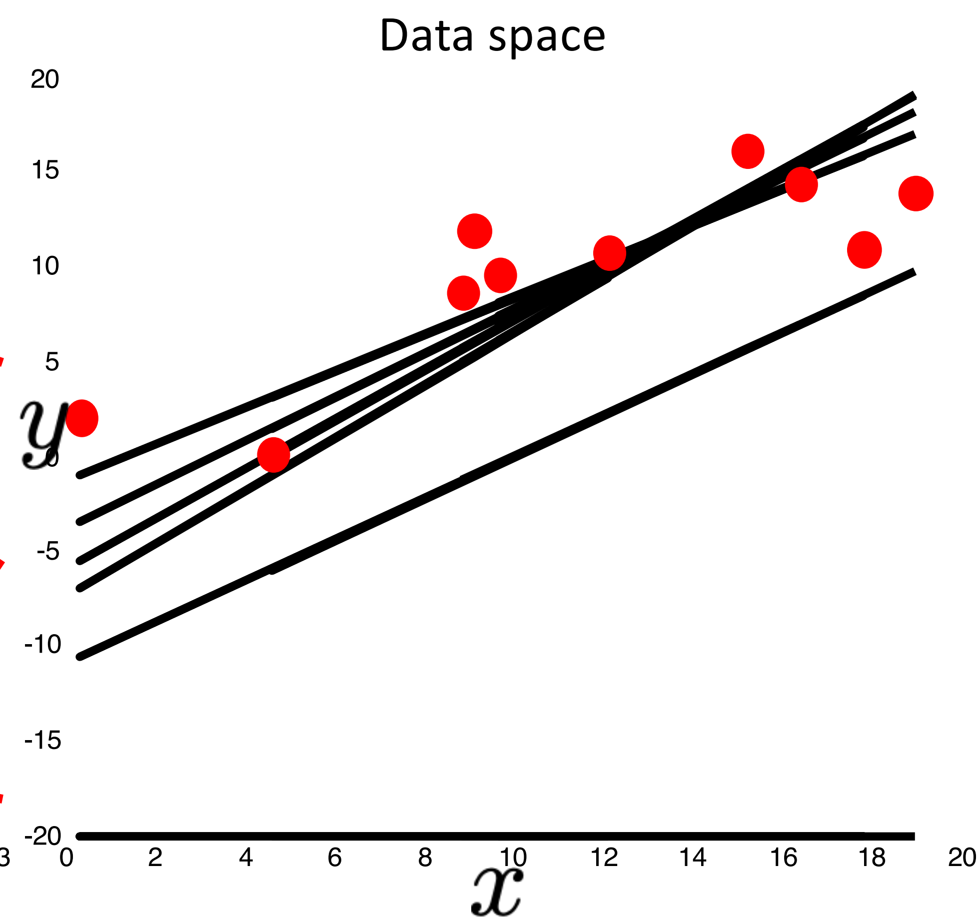
$$\underline{X} = \begin{bmatrix} x_0^{(1)} & \ldots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_0^{(m)} & \ldots & x_n^{(m)} \end{bmatrix}$$

$$\nabla J(\underline{\theta}) = -\frac{2}{m} (\underline{y}^T - \underline{\theta} \underline{X}^T) \cdot \underline{X}$$

```
e = Y - X.dot( theta.T ); # error residual
DJ = - e.dot(X) * 2.0/m  # compute the gradient
theta -= alpha * DJ       # take a step
```

Gradient descent on cost function

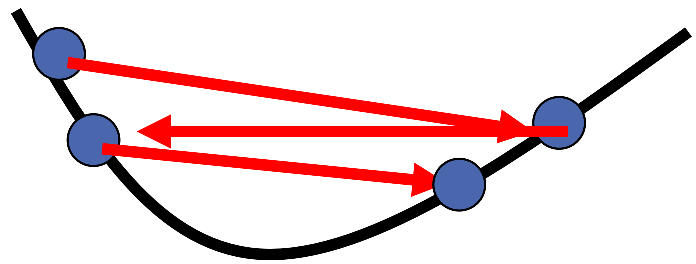




Comments on Gradient Descent

Very general algorithm

- We'll see it many times

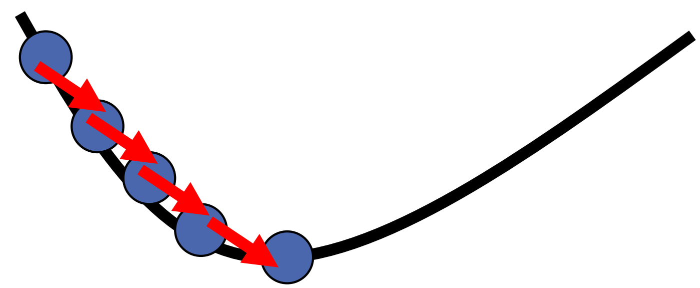


Local minima

- Sensitive to starting point

Step size

- Too large? Too small? Automatic ways to choose?
- May want step size to decrease with iteration
- Common choices:
  - Fixed
  - Linear: C/(iteration)
  - Line search
  - Newton's method



Machine Learning

Gradient Descent

Newton’s Method, Stochastic Gradient Descent

Directly Solving MSE, , L1 Error

Non-linear Regression

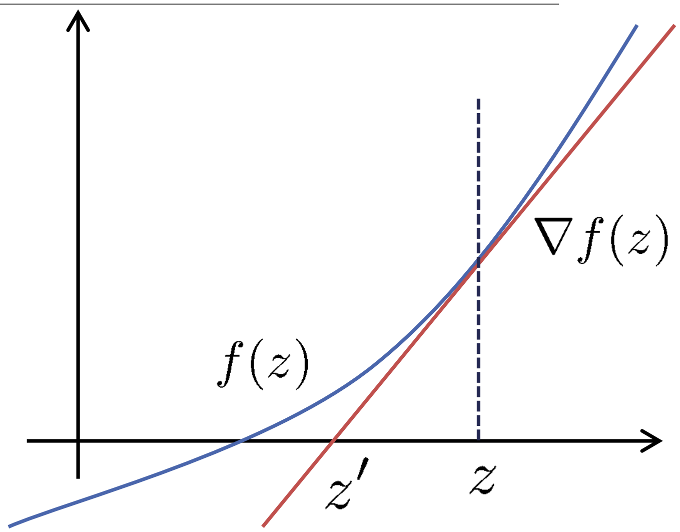
Newton’s method

Want to find the roots of f(z)
- “Root”: value of x for which f(z)=0

Initialize to some point z

Compute tangent at z & where it crosses z-axis

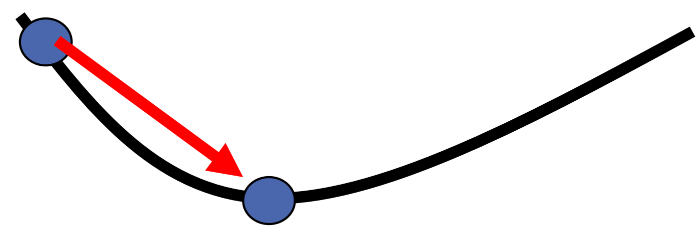
$$\nabla f(z) = \frac{0 - f(z)}{z' - z} \quad \Rightarrow \quad z' = z - \frac{f(z)}{\nabla f(z)}$$



Optimization: find roots of ∇J(θ) (because gradient is zero at the minimum)

Replacing f by ∇J:

$$\nabla\nabla J(\theta) = \frac{0 - \nabla J(\theta)}{\theta' - \theta} \quad \Rightarrow \quad \theta' = \theta - \frac{\nabla J(\theta)}{\nabla\nabla J(\theta)}$$

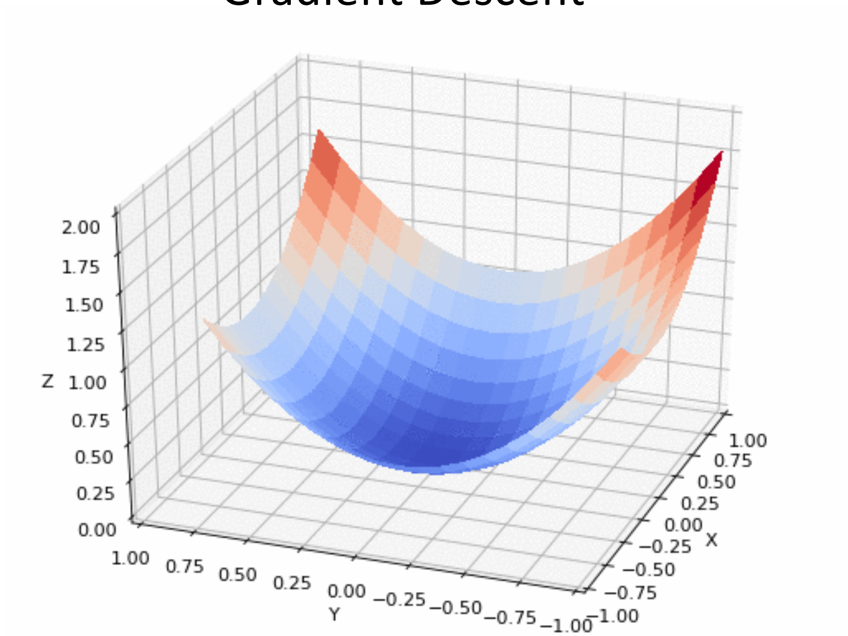


- Does not always converge; sometimes unstable
- If converges, usually very fast
- Works well for smooth, non-pathological functions, locally quadratic
- For n large, may be computationally hard: $O(n^2)$ storage, $O(n^3)$ time

Stochastic / Online Gradient Descent

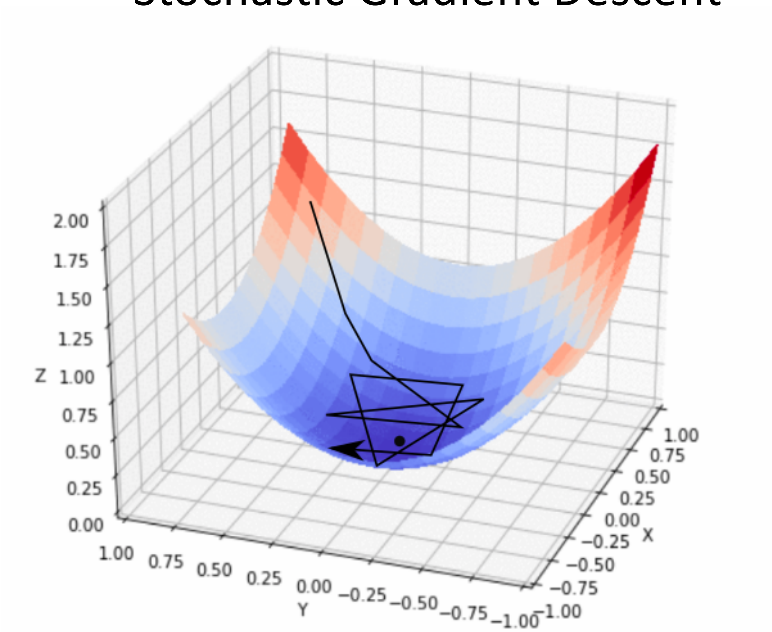
Goal: estimate the gradient *cheaply* based on sub-sampled data points

Gradient Descent



Straight path; every step is expensive

Stochastic Gradient Descent



Zig-zag path; every step is cheap

https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/

Stochastic / Online Gradient Descent

Goal: estimate the gradient *cheaply* based on sub-sampled data points

MSE
$$J(\underline{\theta}) = \frac{1}{m}\sum_j J_j(\underline{\theta}), \qquad J_j(\underline{\theta}) = (y^{(j)} - \underline{\theta}\cdot \underline{x}^{(j)^T})^2$$

Gradient (sum over all datapoints):

$$\nabla J(\theta) = \frac{1}{m}\sum_{j=1}^{m} \nabla J_j(\theta)$$

Stochastic gradient (selects datapoint j at random):

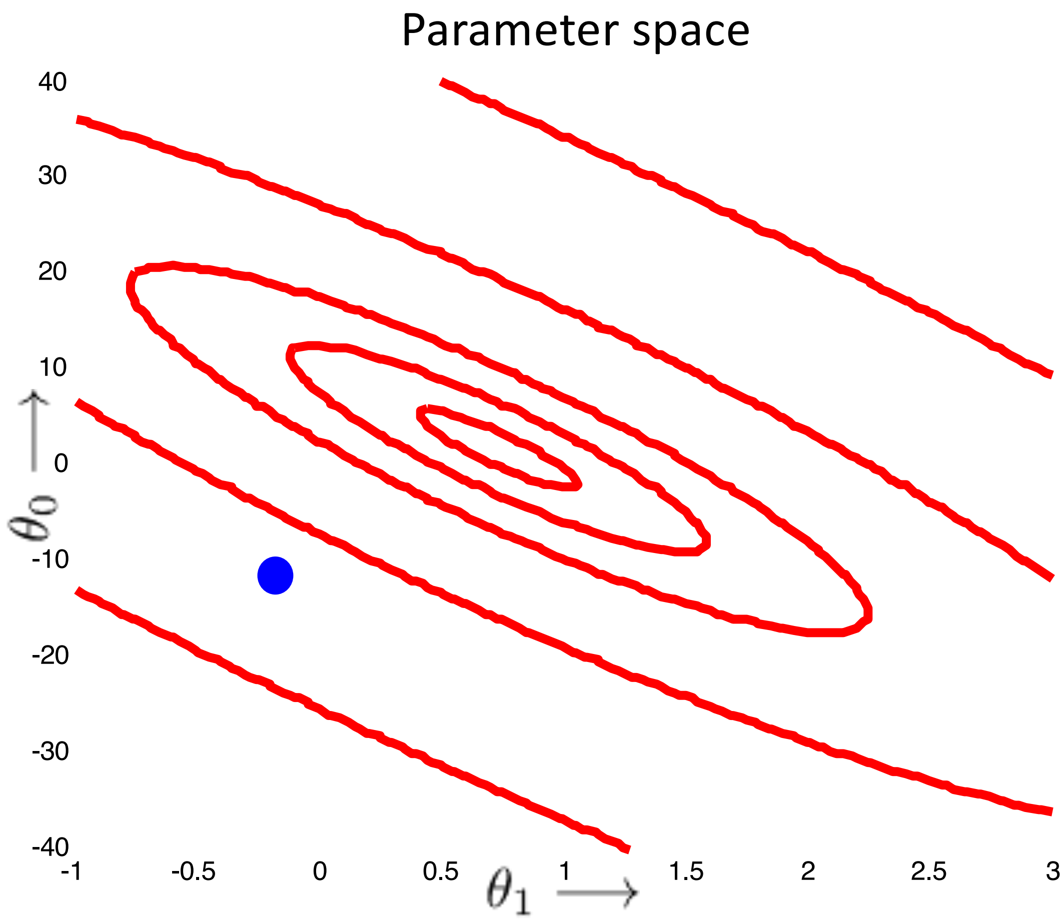
$$\hat{\nabla} J(\theta) = \nabla J_j(\theta), \quad \text{where } j \sim Uniform\{1, \ldots, m\}$$

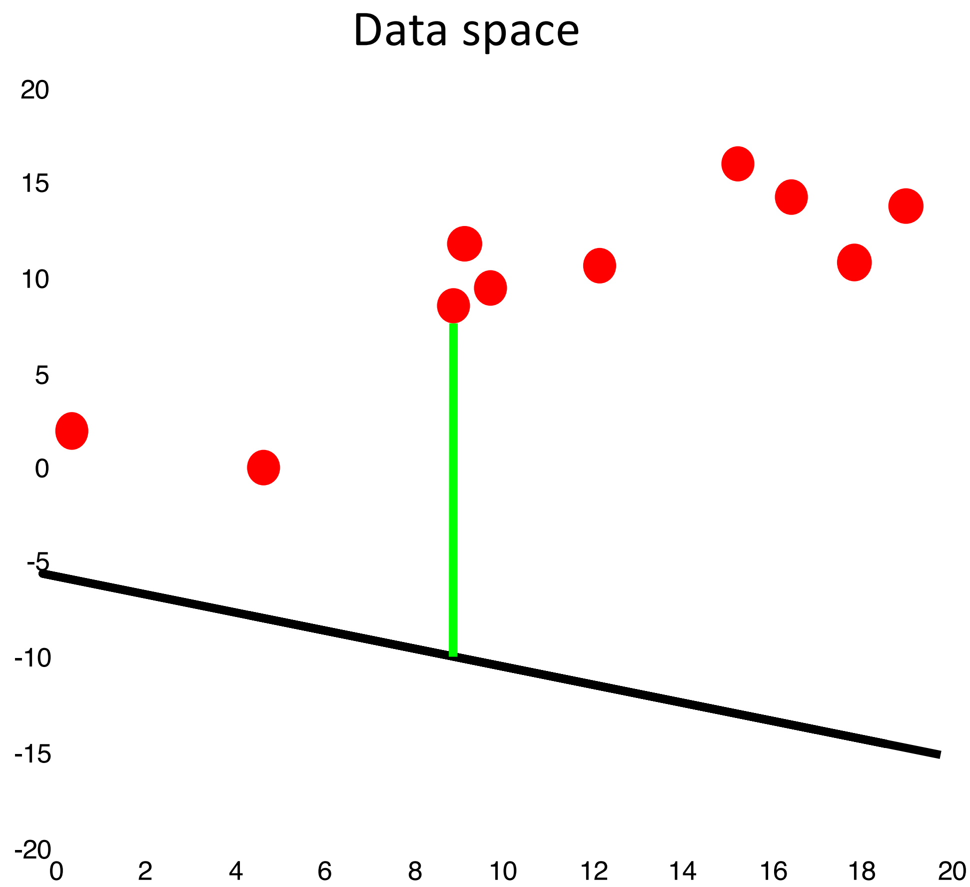
The "average" direction of the stochastic gradient is the (true) gradient:

$$E\left[\hat{\nabla} J(\theta)\right] = \nabla J(\theta) \quad \text{(average over the data)}$$

Stochastic gradient descent

Update based on each datum at a time

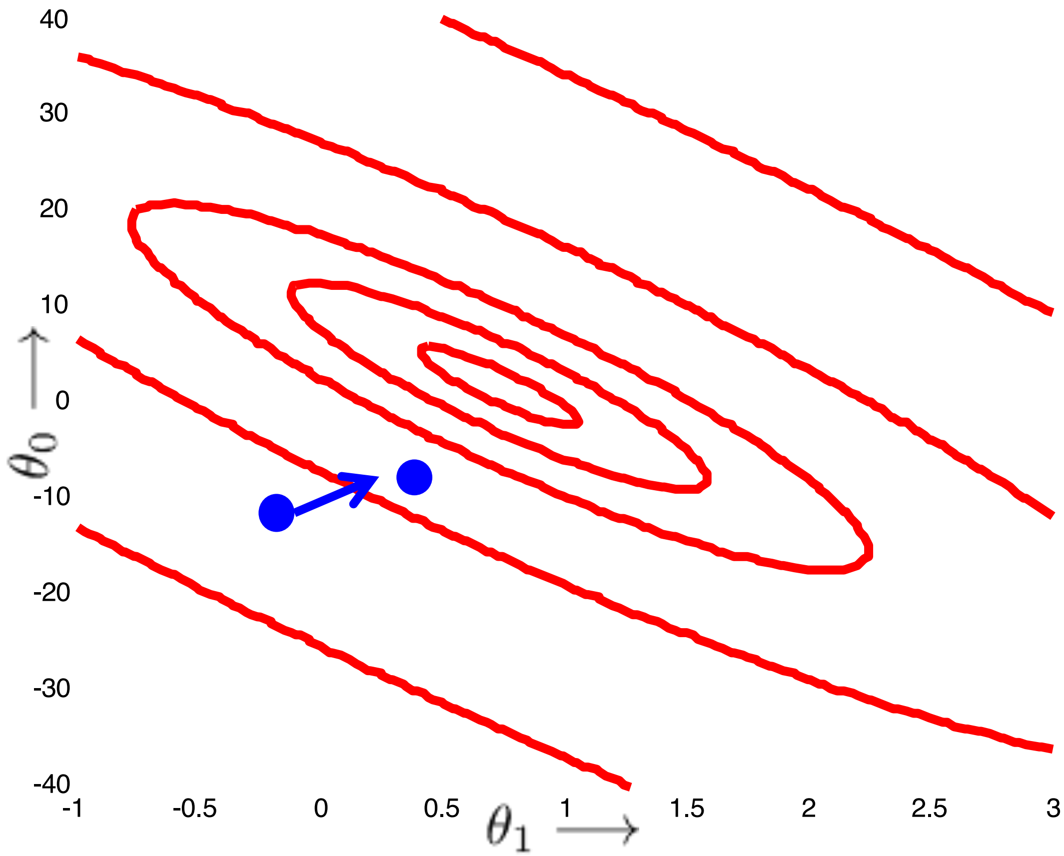




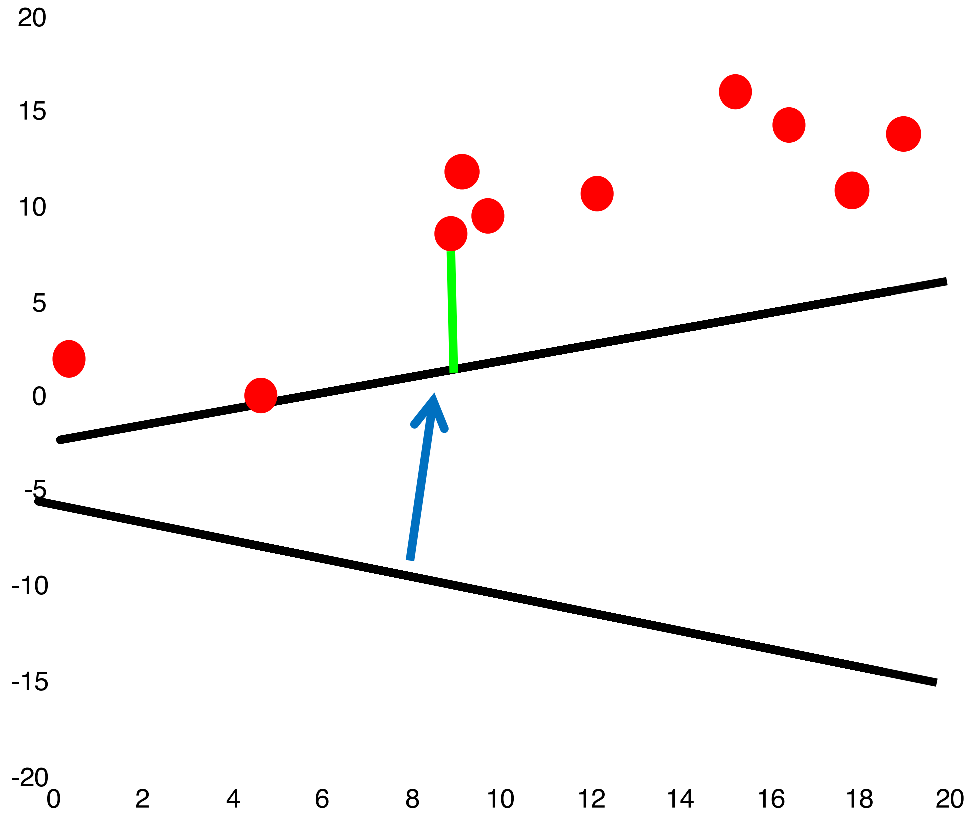
Stochastic gradient descent

Update based on each datum at a time

Parameter space



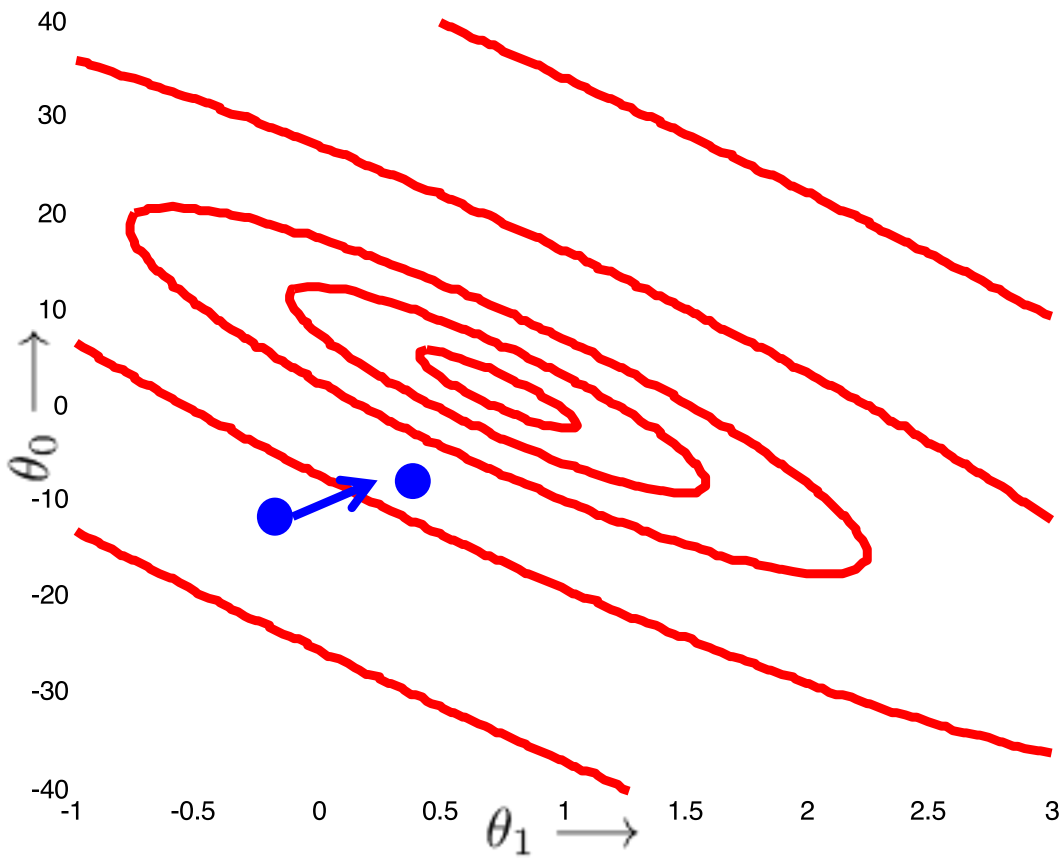
Data space



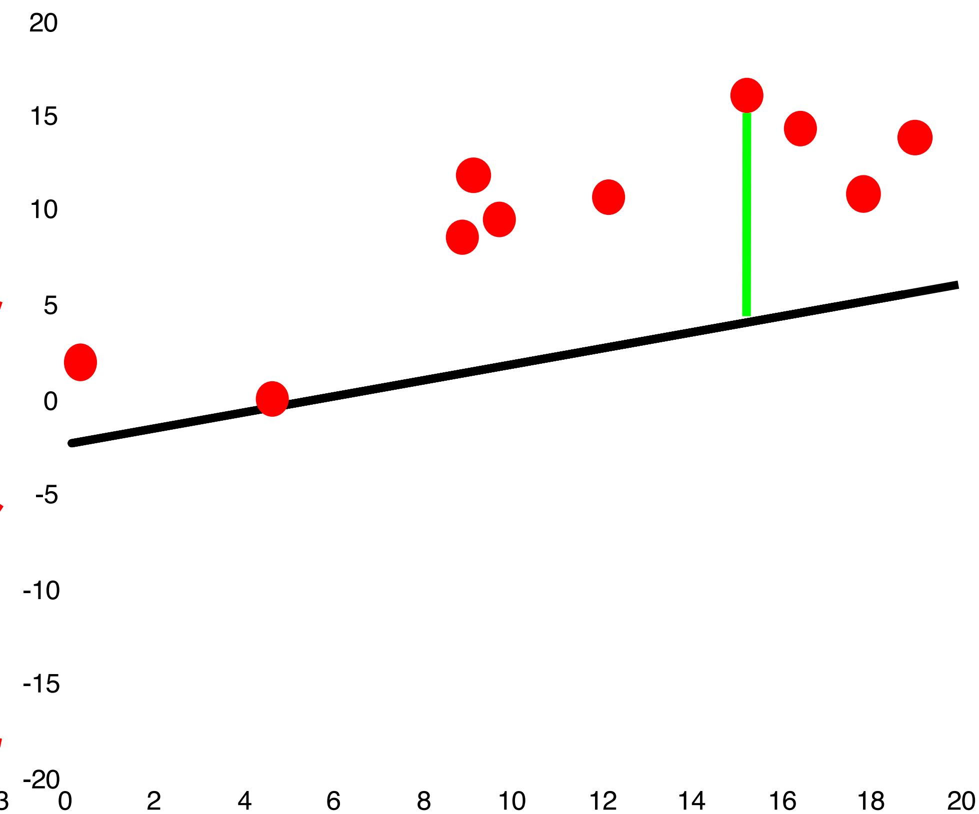
Stochastic gradient descent

Update based on each datum at a time

Parameter space

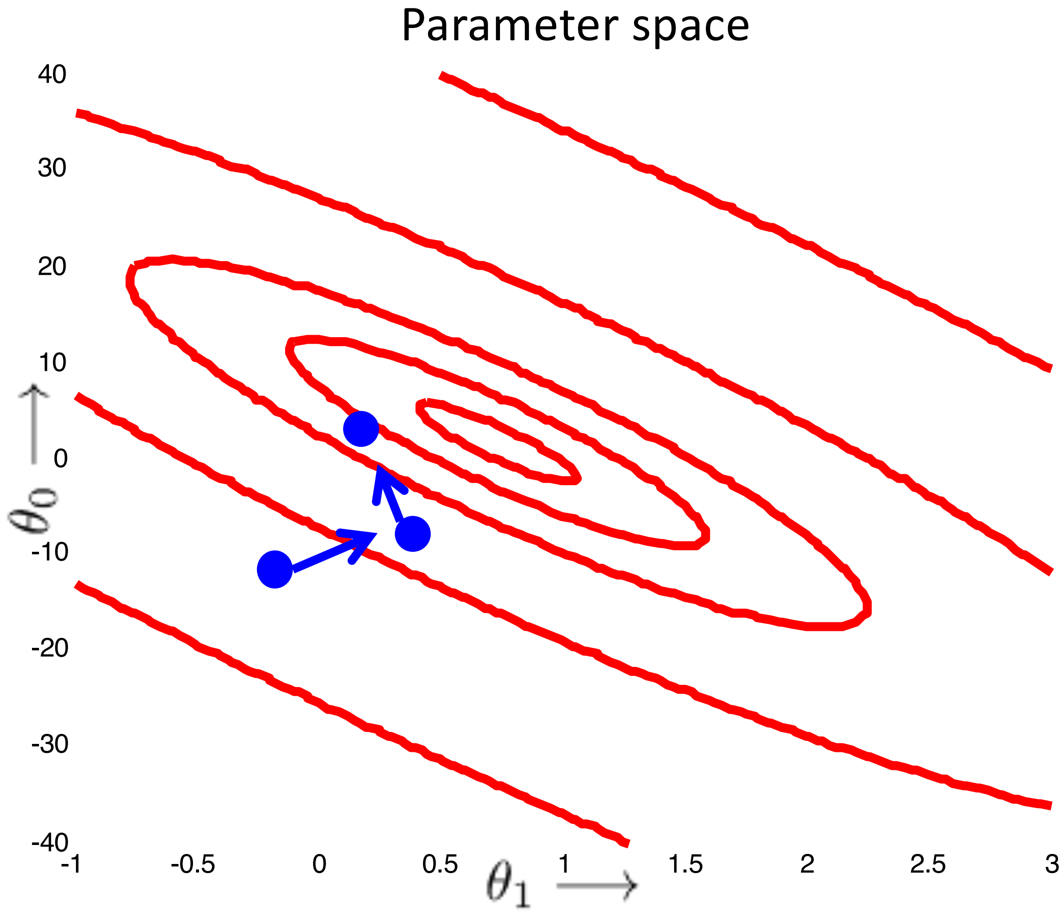


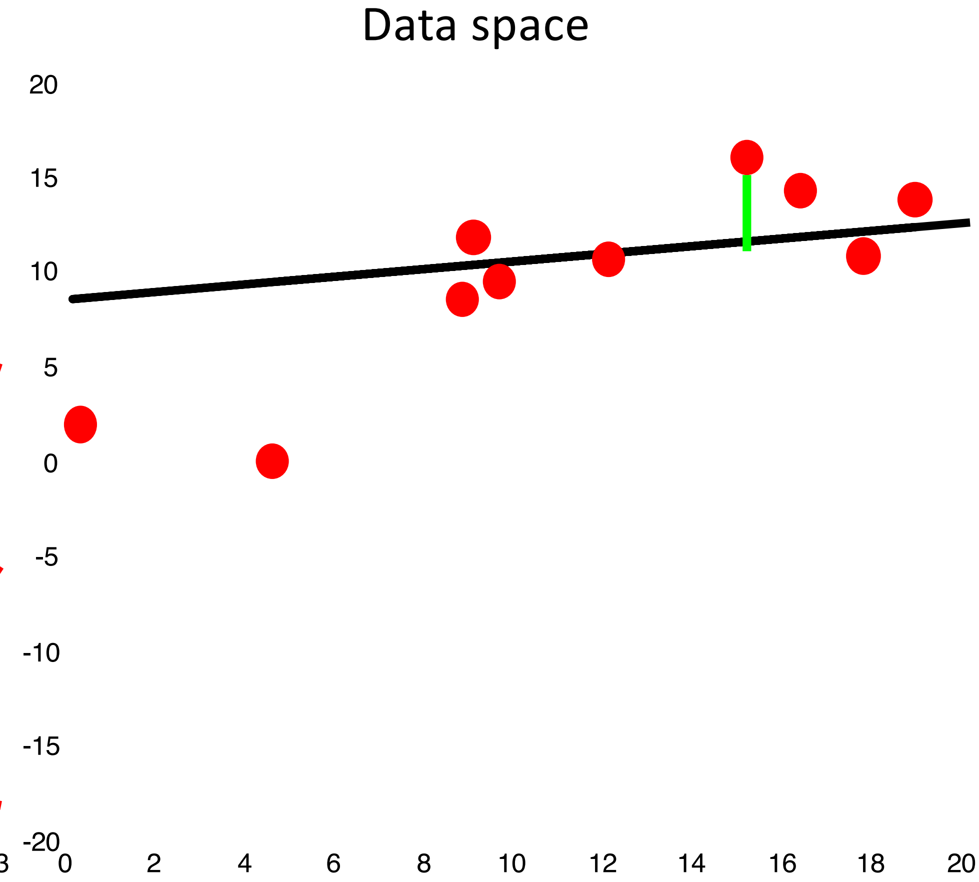
Data space



Stochastic gradient descent

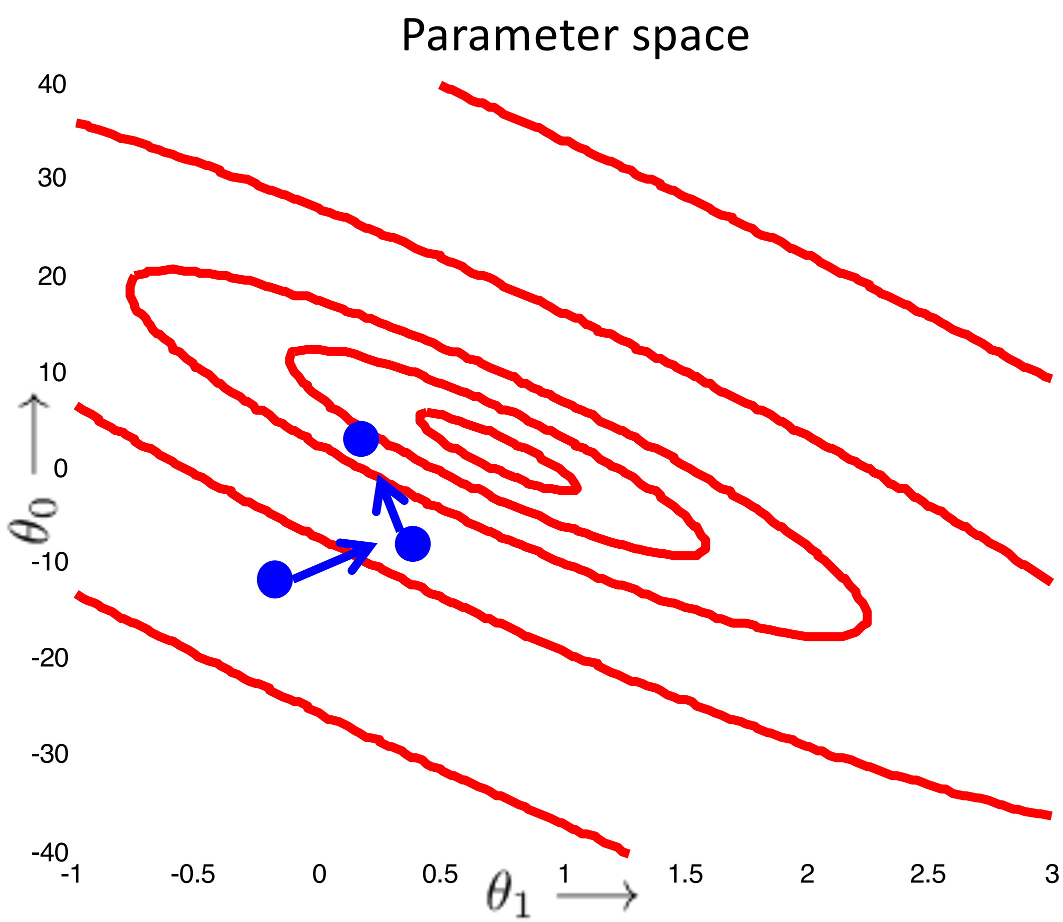
Update based on each datum at a time

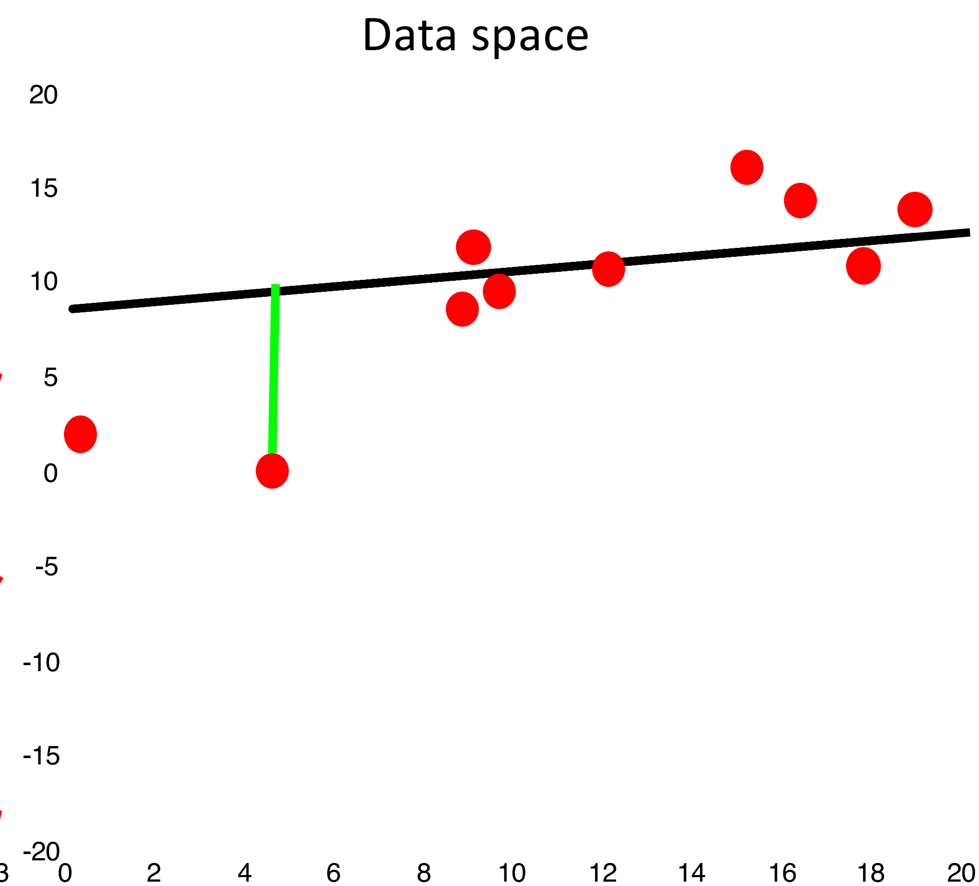




Stochastic gradient descent

Update based on each datum at a time

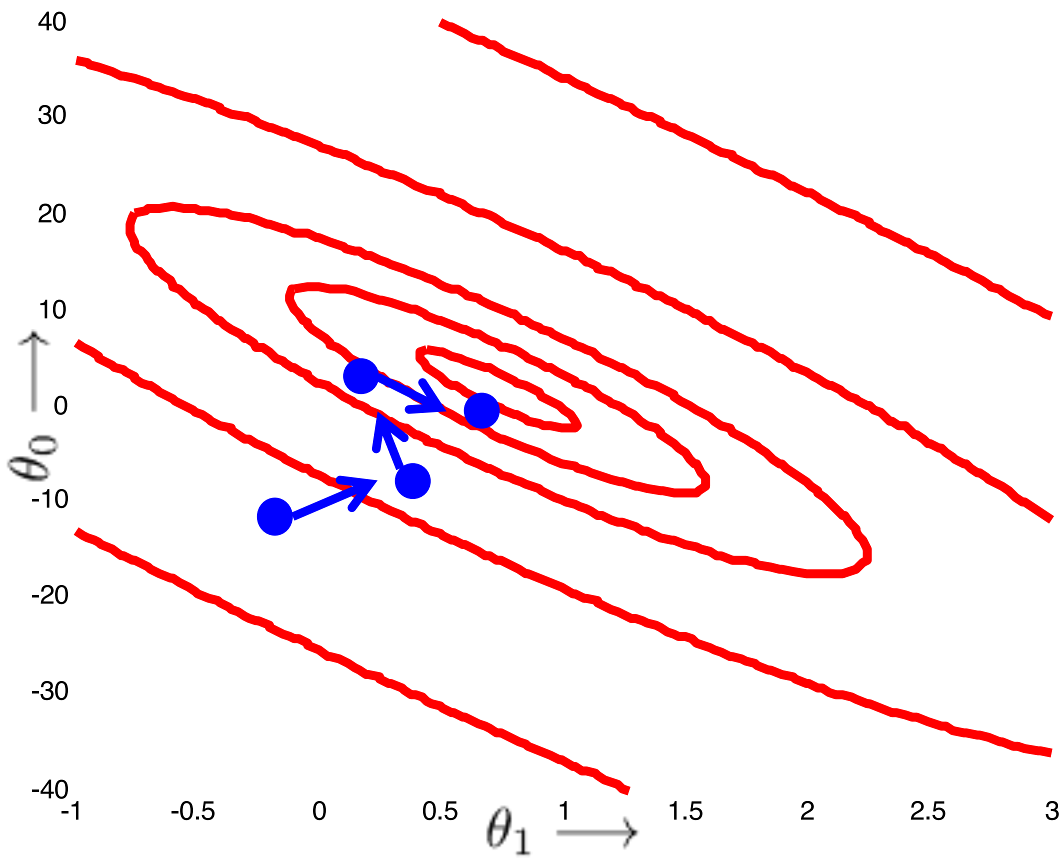




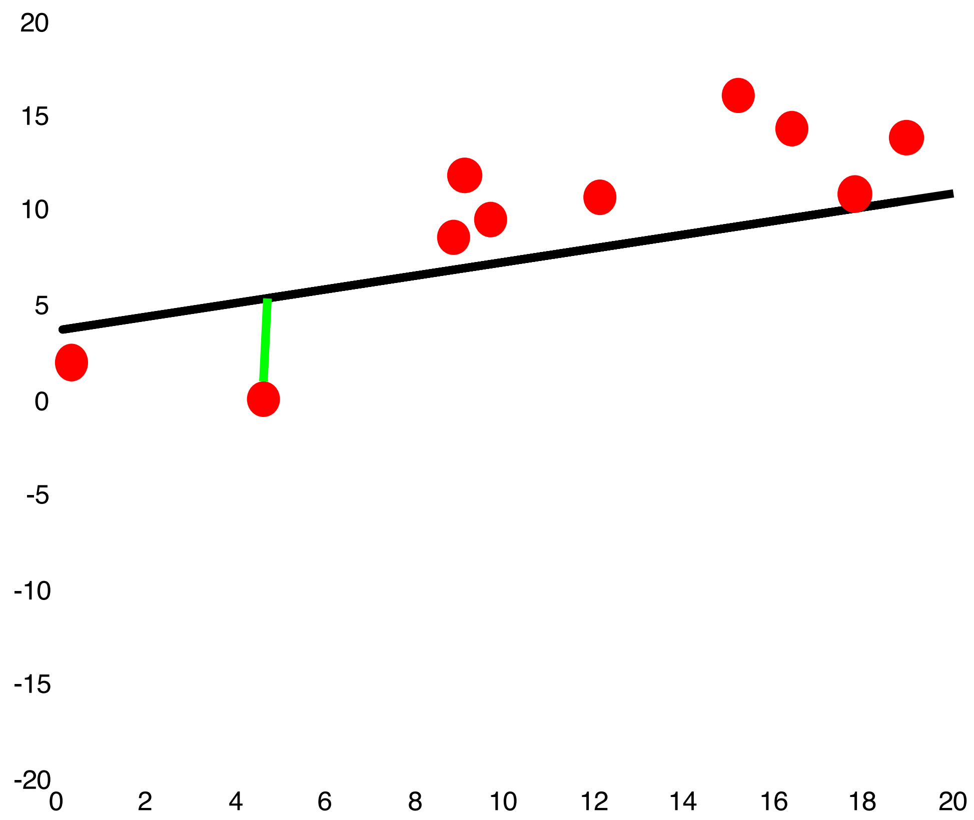
Stochastic gradient descent

Update based on each datum at a time

Parameter space



Data space



Stochastic gradient descent

$$J_j(\underline{\theta}) = (y^{(j)} - \underline{\theta} \cdot \underline{x}^{(j)^T})^2$$

$$\nabla J_j(\underline{\theta}) = -2(y^{(j)} - \underline{\theta} \cdot \underline{x}^{(j)^T}) \cdot [x_0^{(j)} x_1^{(j)} \ldots]$$

```
Initialize θ,
shuffle data set
Do {
 for j=1:m
   θ ← θ - α∇θJj(θ)
} while (not done)
```

Benefits

- Lots of data = many more updates per pass
- Computationally faster
- Arguably the most important optimization algorithm nowadays

Drawbacks

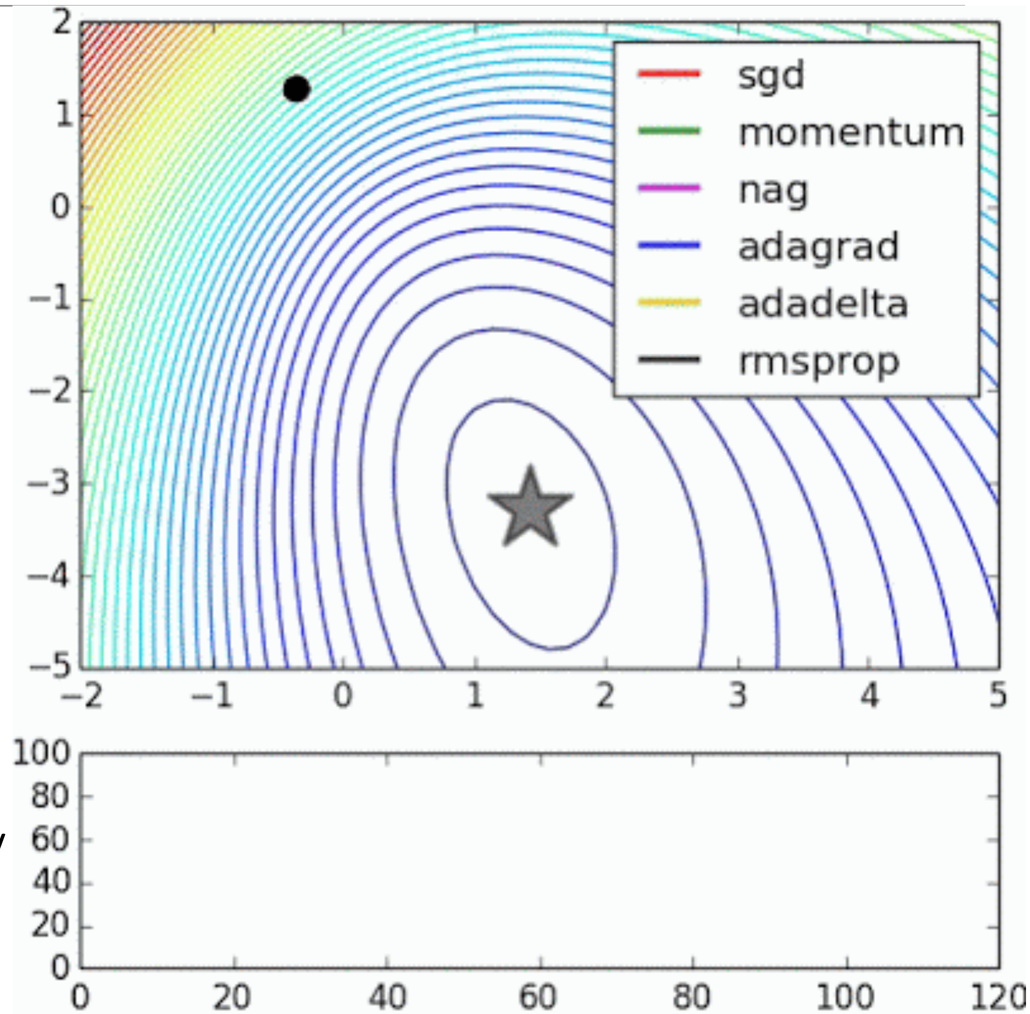
- No longer strictly "descent"
- Stopping conditions may be harder to evaluate
  (Can use "running estimates" of J(.), etc.)

Related:

- mini-batch updates, etc.

Advanced SGD algorithms

- Adding a momentum term
  - Keeps memory of “velocity”
  - Smoothens stochastic path
- Preconditioning
  - Pre-multiply gradient, e.g., with inverse Hessian (diagonal approximation)
  - Makes optimization problem “better behaved”



Blog Post:
https://ruder.io/optimizing-gradient-descent/

Source: http://www.denizyuret.com/

Machine Learning

Gradient Descent

Newton's Method, Stochastic Gradient Descent

Directly Solving MSE, L1 Error

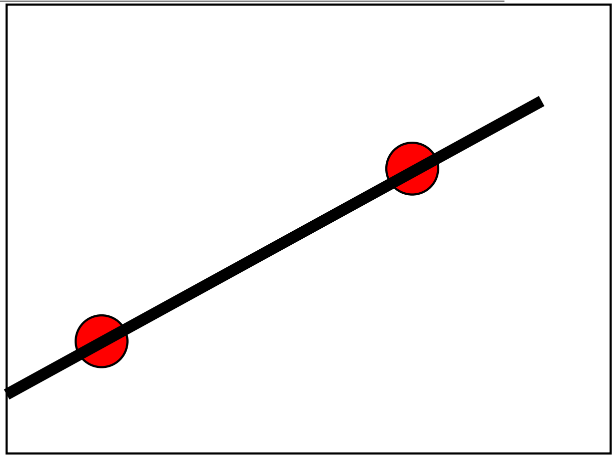
Non-linear Regression

MSE Minimum

Consider a simple problem

- One feature, two data points
- Two unknowns: θ_0, θ_1
- Two equations:

$$y^{(1)} = \theta_0 + \theta_1 x^{(1)}$$
$$y^{(2)} = \theta_0 + \theta_1 x^{(2)}$$



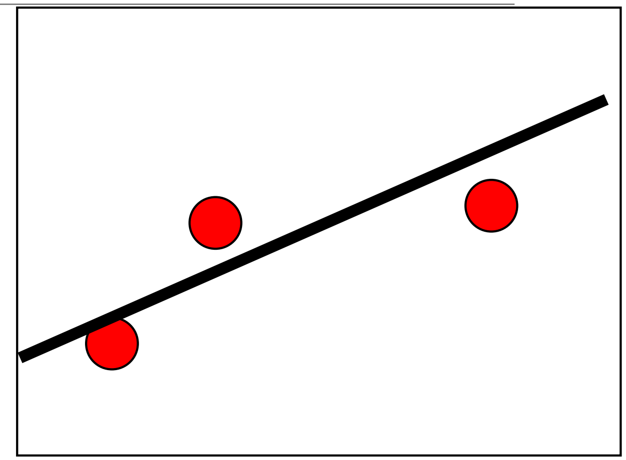
- Can solve this system directly:

$$\underline{y}^T = \underline{\theta}\,\underline{X}^T \qquad \Rightarrow \qquad \underline{\hat{\theta}} = y^T (\underline{X}^T)^{-1}$$

- However, most of the time, m > n
  - There may be no linear function that hits all the data exactly
  - Instead, solve directly for minimum of MSE function

MSE Minimum

$$\nabla J(\underline{\theta}) = -\frac{2}{m}(\underline{y}^T - \underline{\theta X^T}) \cdot \underline{X} \quad = \quad \underline{0}$$

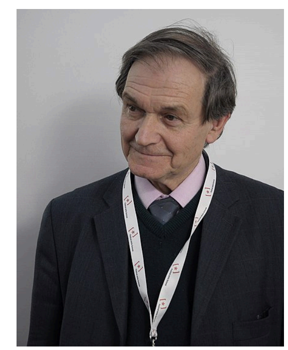


Reordering, we have

$$\underline{y}^T \underline{X} - \underline{\theta X^T} \cdot \underline{X} \quad = \quad \underline{0}$$

$$\underline{y}^T \underline{X} = \underline{\theta X^T} \cdot \underline{X}$$

$$\underline{\theta} \quad = \quad \underline{y}^T \underline{X}(\underline{X}^T \underline{X})^{-1}$$



R. Penrose
Nobel Prize
2020

- $X (X^T X)^{-1}$ is the Moore-Penrose "pseudo-inverse"
- If X^T is square and independent, this is the inverse
- If m > n: overdetermined; gives minimum MSE fit

Python MSE

This is easy to solve in Python / NumPy…

$$\underline{\theta} \quad = \quad \underline{y}^T \, \underline{X}(\underline{X}^T \, \underline{X})^{-1}$$

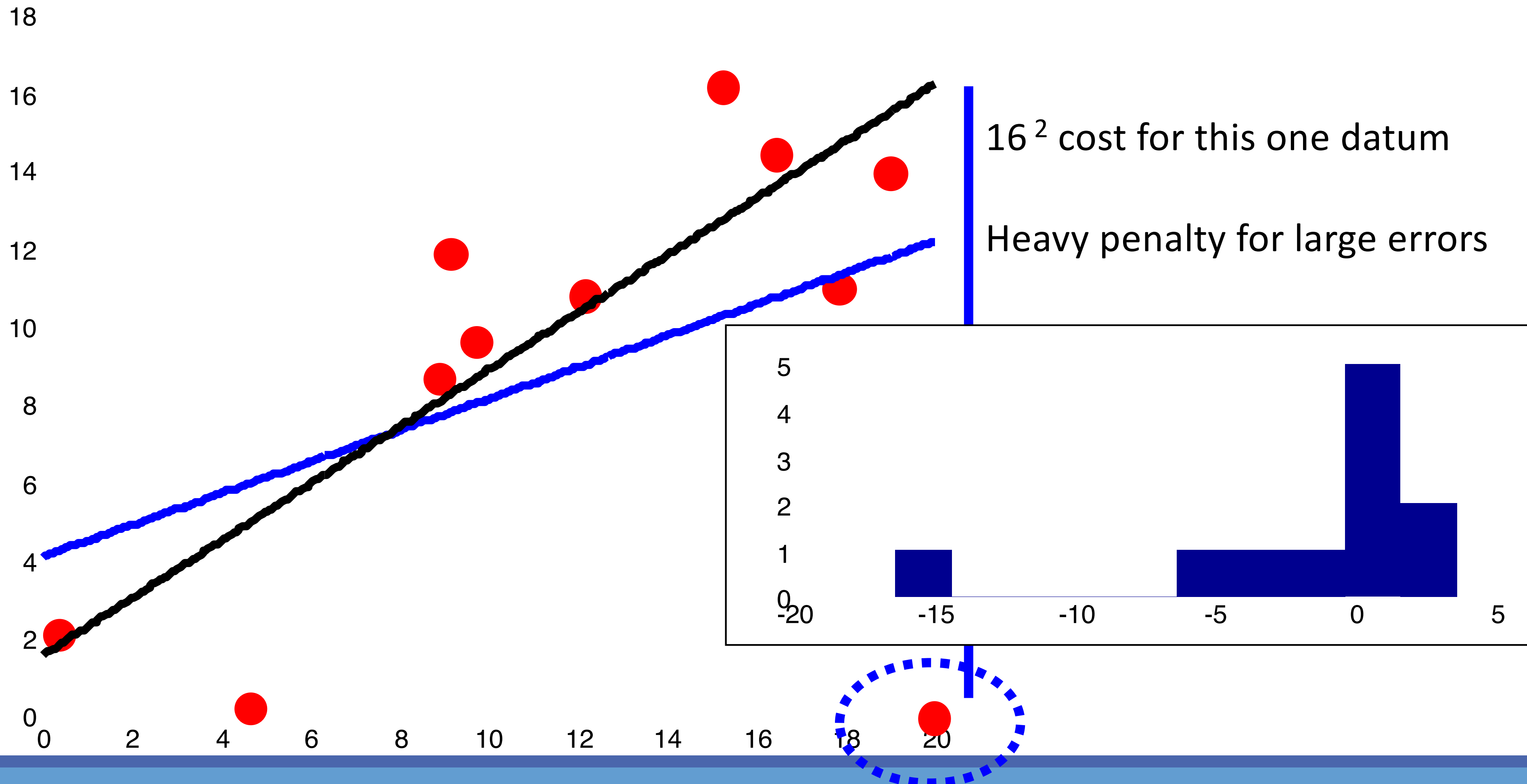
```
#  y = np.matrix( [[y1], … , [ym]] )
#  X = np.matrix( [[x1_0 … x1_n], [x2_0 … x2_n], …] )
```

```
# Solution 1: "manual"
   th = y.T * X * np.linalg.inv(X.T * X);
```

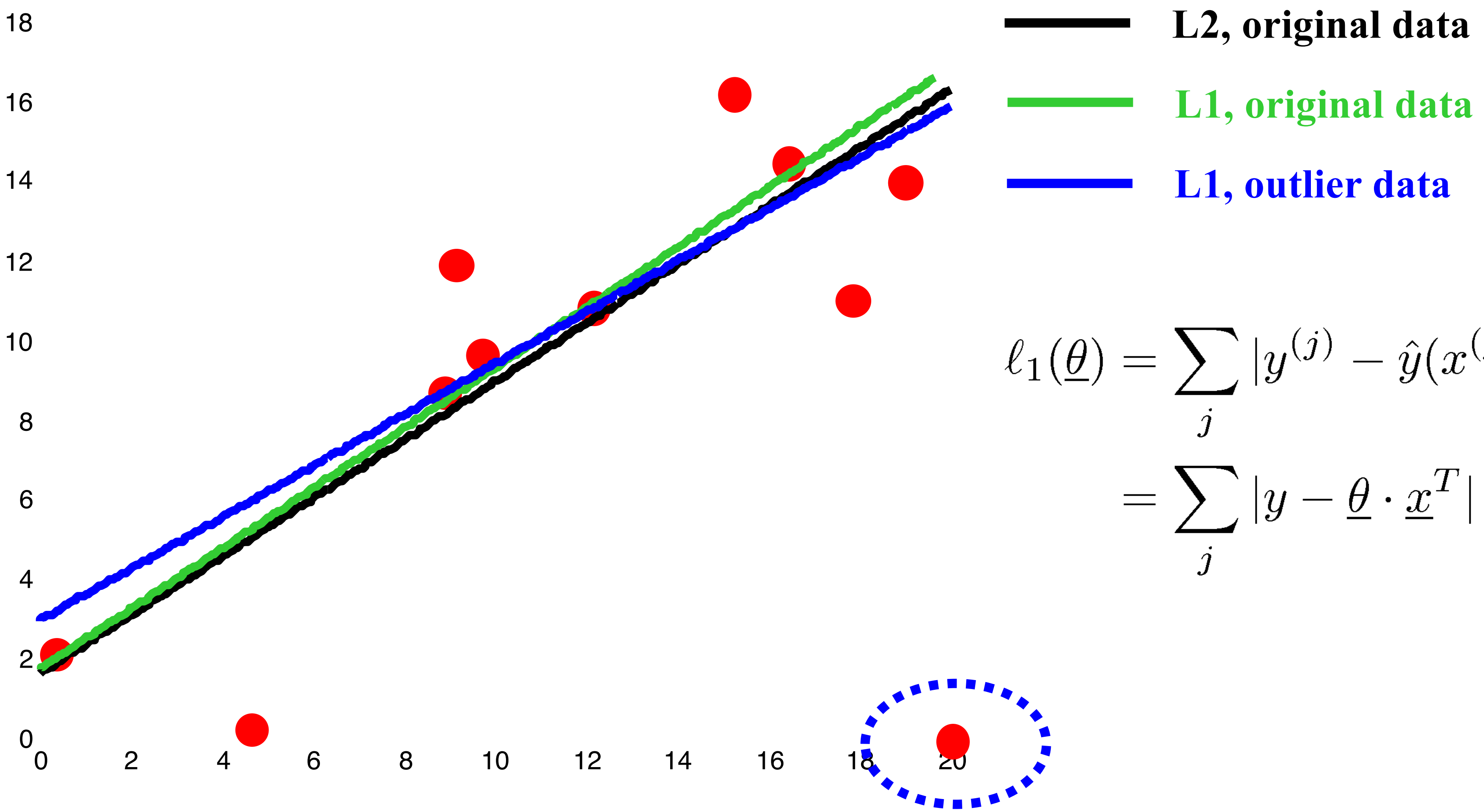
```
# Solution 2: "least squares solve"
   th = np.linalg.lstsq(X, Y);
```

Effects of MSE choice

Sensitivity to outliers



L1 error: Mean Absolute Error



$$\ell_1(\underline{\theta}) = \sum_j |y^{(j)} - \hat{y}(x^{(j)})|$$

$$= \sum_j |y - \underline{\theta} \cdot \underline{x}^T|$$

Cost functions for regression

$$\ell_2 \;:\; (y - \hat{y})^2$$ **(MSE)**

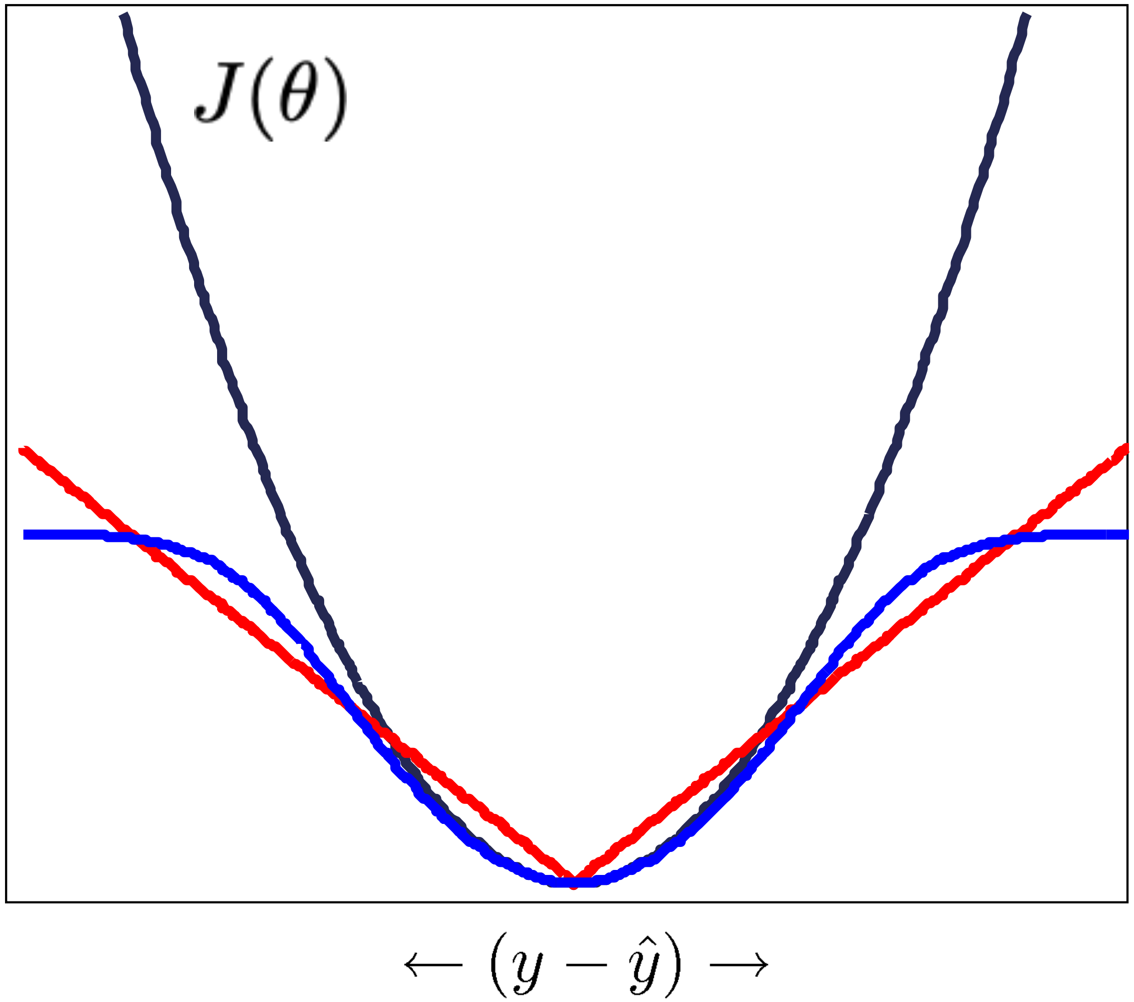
$$\ell_1 \;:\; |y - \hat{y}|$$ **(MAE)**

Something else entirely…

$$c - \log(\exp(-(y - \hat{y})^2) + c)$$

(???)

Arbitrary functions cannot be solved in closed form
- use gradient descent



Machine Learning

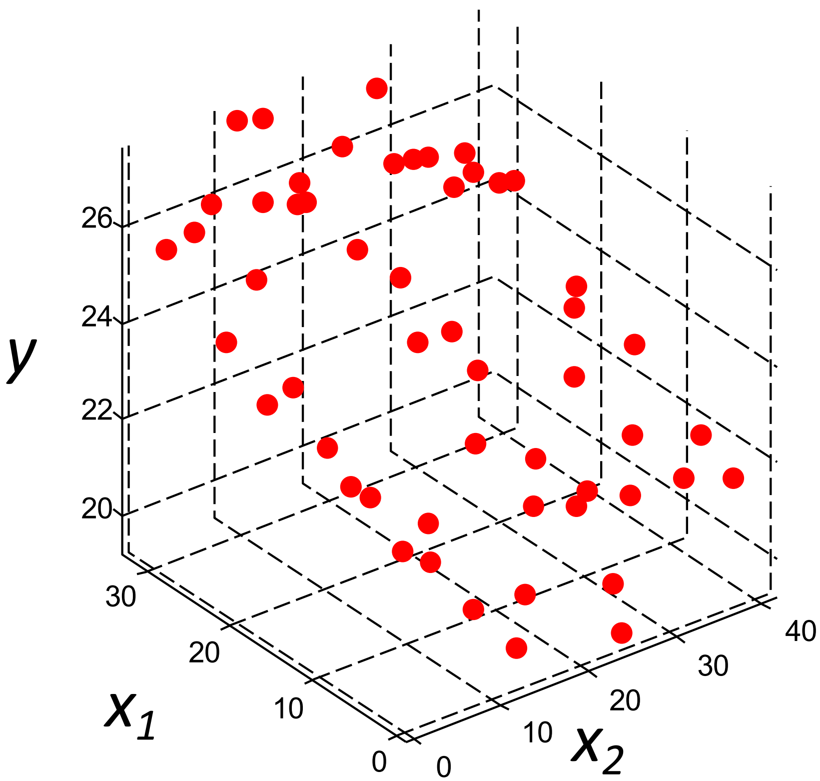
Gradient Descent

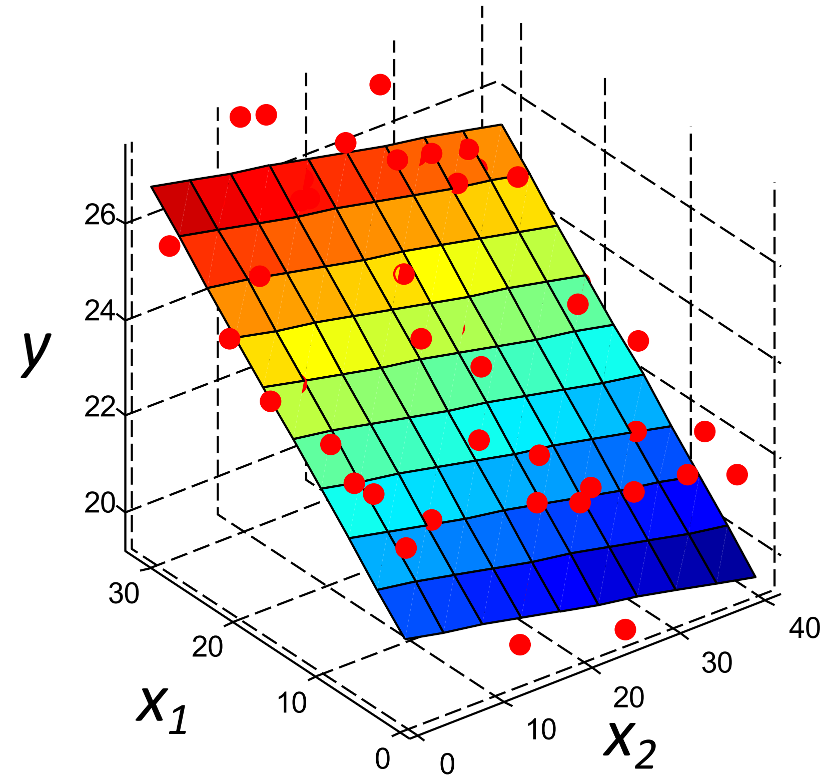
Newton's Method, Stochastic Gradient Descent

Directly Solving MSE, L1 Error

Non-linear Regression

Linear Regression in higher dimensions





$$\hat{y}(x) = \underline{\theta} \cdot \underline{x}^T$$

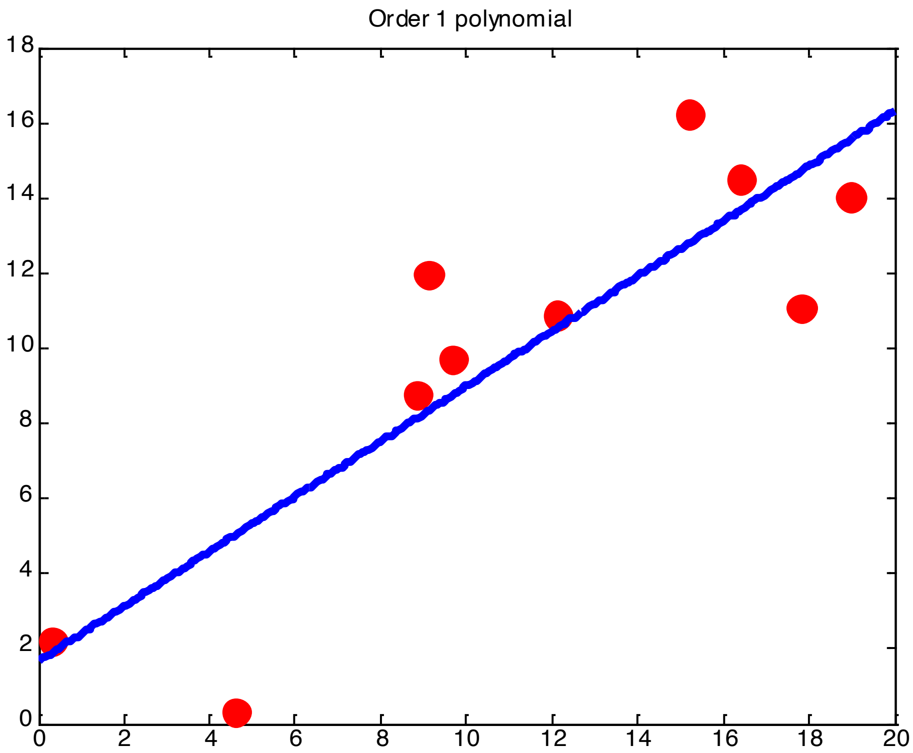
$$\underline{\theta} = [\theta_0 \ \theta_1 \ \theta_2]$$

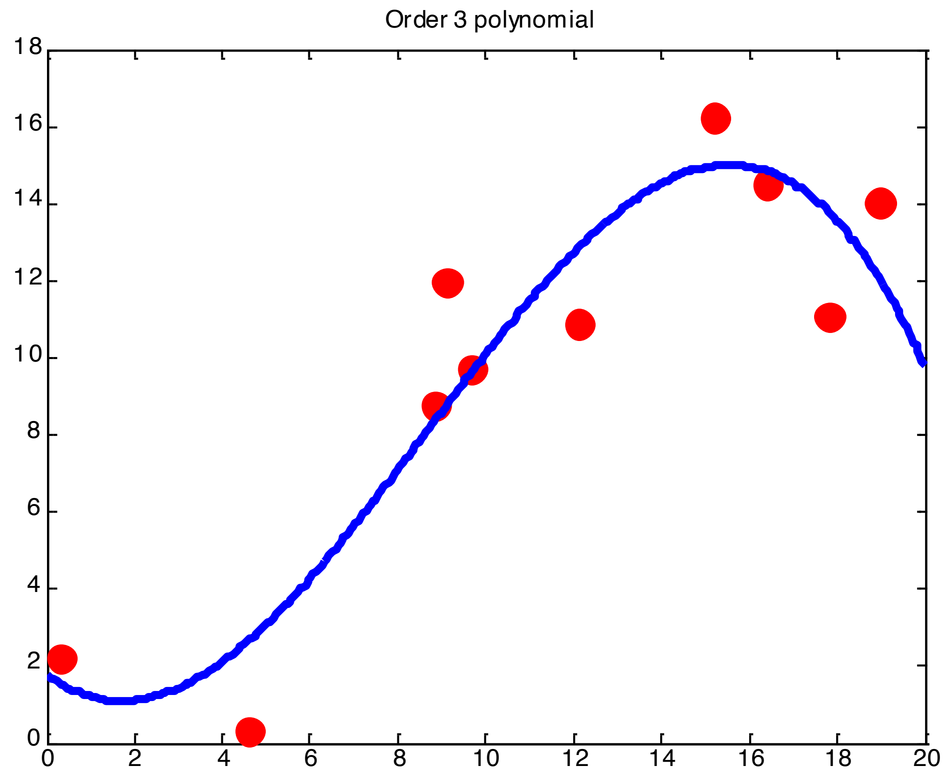
$$\underline{x} = [1 \ x_1 \ x_2]$$

Nonlinear functions

Sometimes we are interested in fitting non-linear functions

- Ex: higher-order polynomials





Nonlinear functions

Single feature x, predict target y:

$$D = \{(x^{(j)}, y^{(j)})\}$$

$$\Downarrow \quad \text{Add features:}$$

$$D = \{([x^{(j)}, (x^{(j)})^2, (x^{(j)})^3], y^{(j)})\}$$

$$\hat{y}(x) = \theta_0 + \theta_1\, x + \theta_2\, x^2 + \theta_3\, x^3$$

$$\Downarrow$$

$$\hat{y}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$$

Linear regression in new features

Polynomial regression in *low* dimensions ↔ *linear* regression in *high* dimensions

$$\Phi(x) = \left[1\,,\, x\,,\, x^2\,,\, x^3\,,\, \ldots\right]$$

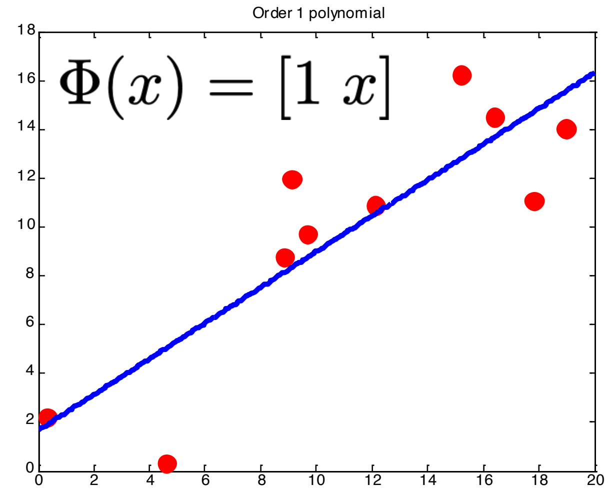
$$\hat{y}(x) = \underline{\theta} \cdot \Phi(x)$$

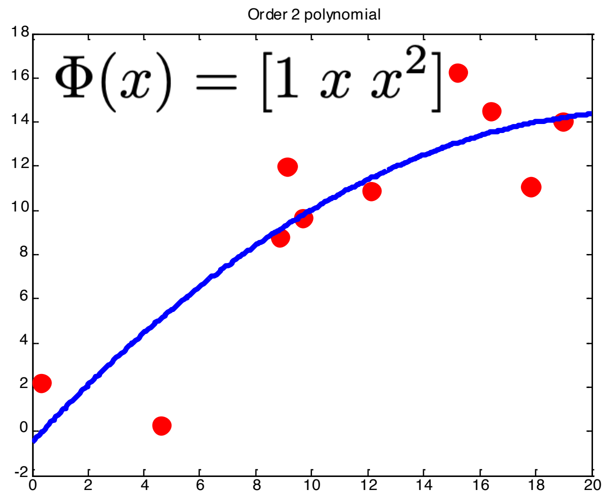
Sometimes useful to think of “feature transform”

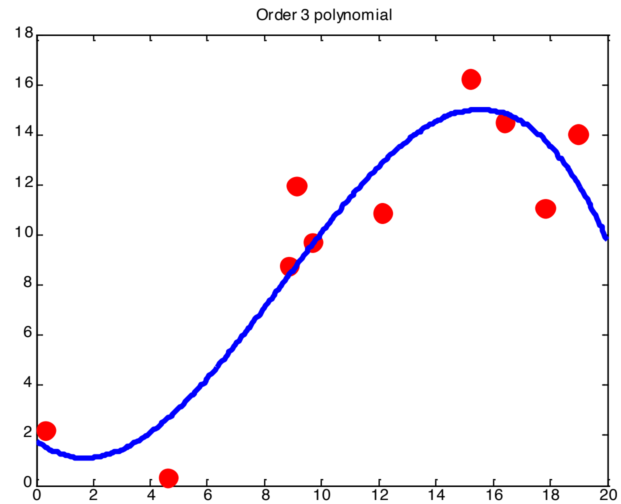
Higher-order polynomials

Fit in the same way

More "features" $\theta \cdot \Phi(x)$







Features

In general, can use any features we think are useful

Instead of collecting more information about datum, just apply nonlinear transformation to existing features

Polynomial functions

- Features [1, x, x2, x3, …]

Other functions

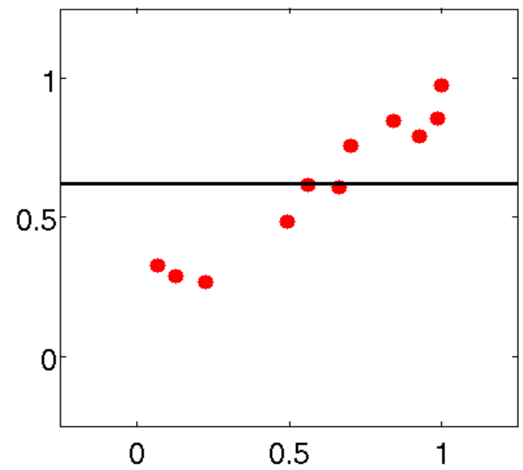
- 1/x, sqrt(x), x1 * x2, …

"Linear regression" = linear in the parameters

- Features can be made as complex as we want!

Higher-order polynomials

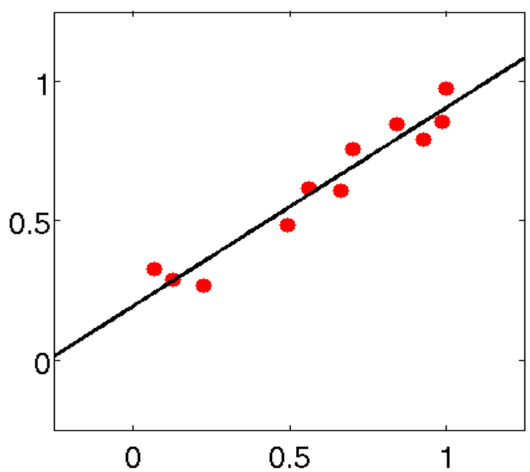
When should we stop adding more features?

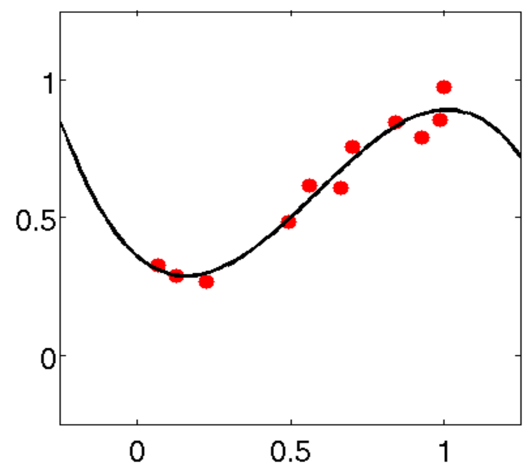


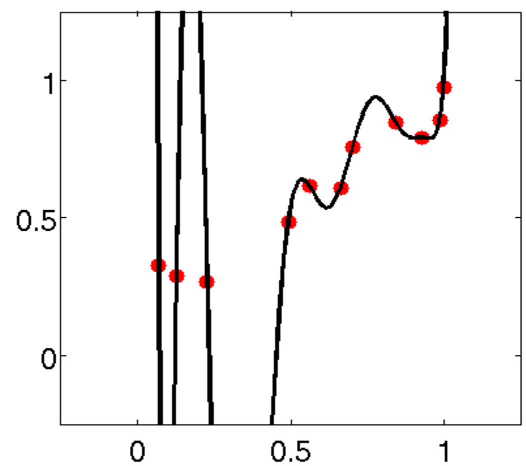
"Nested" hypotheses

- 2nd order more general than 1st,
- 3rd order " " than 2nd, …

Fits the observed data better



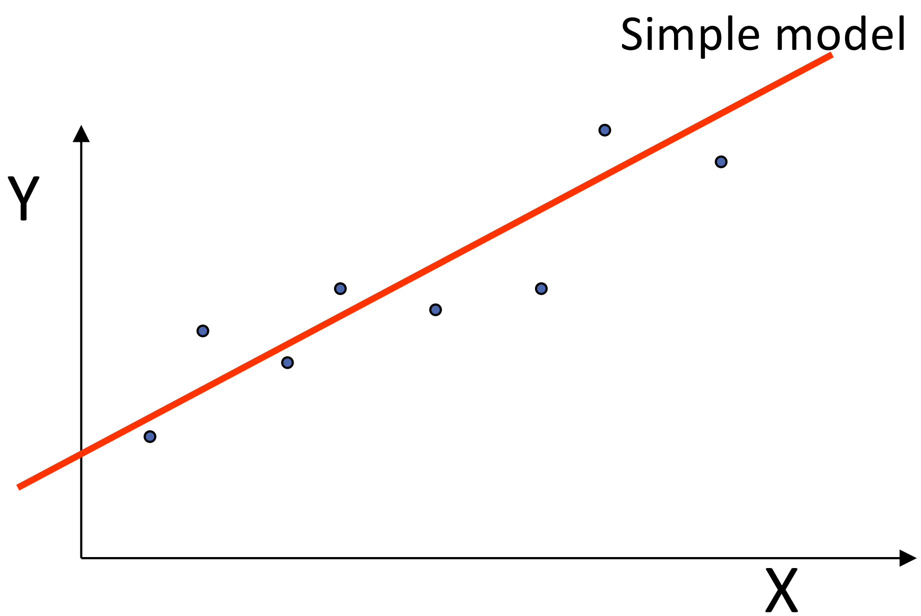


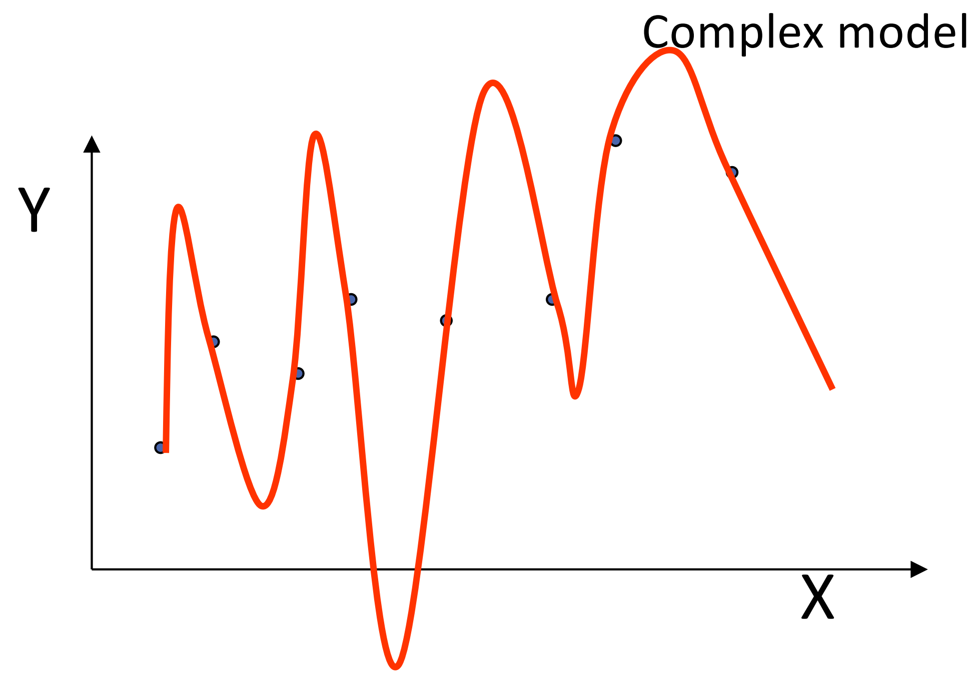


Overfitting and complexity

More complex models will always fit the training data better

But they may "overfit" the training data, learning complex relationships that are not really present





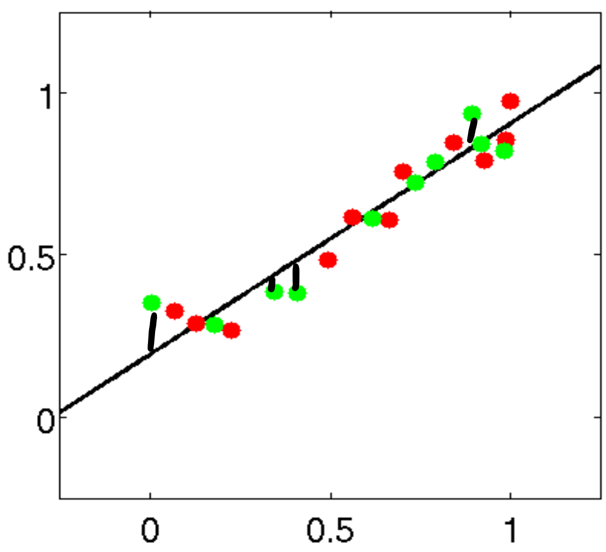
Test data

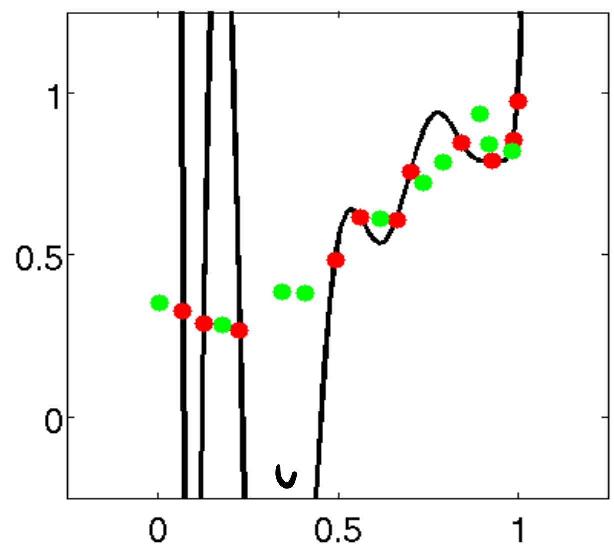
After training the model

Go out and get more data from the world

- New observations (x,y)

How well does our model perform?





Training versus test error

Plot MSE as a function of model complexity

- Polynomial order

Decreases

- More complex function fits training data better

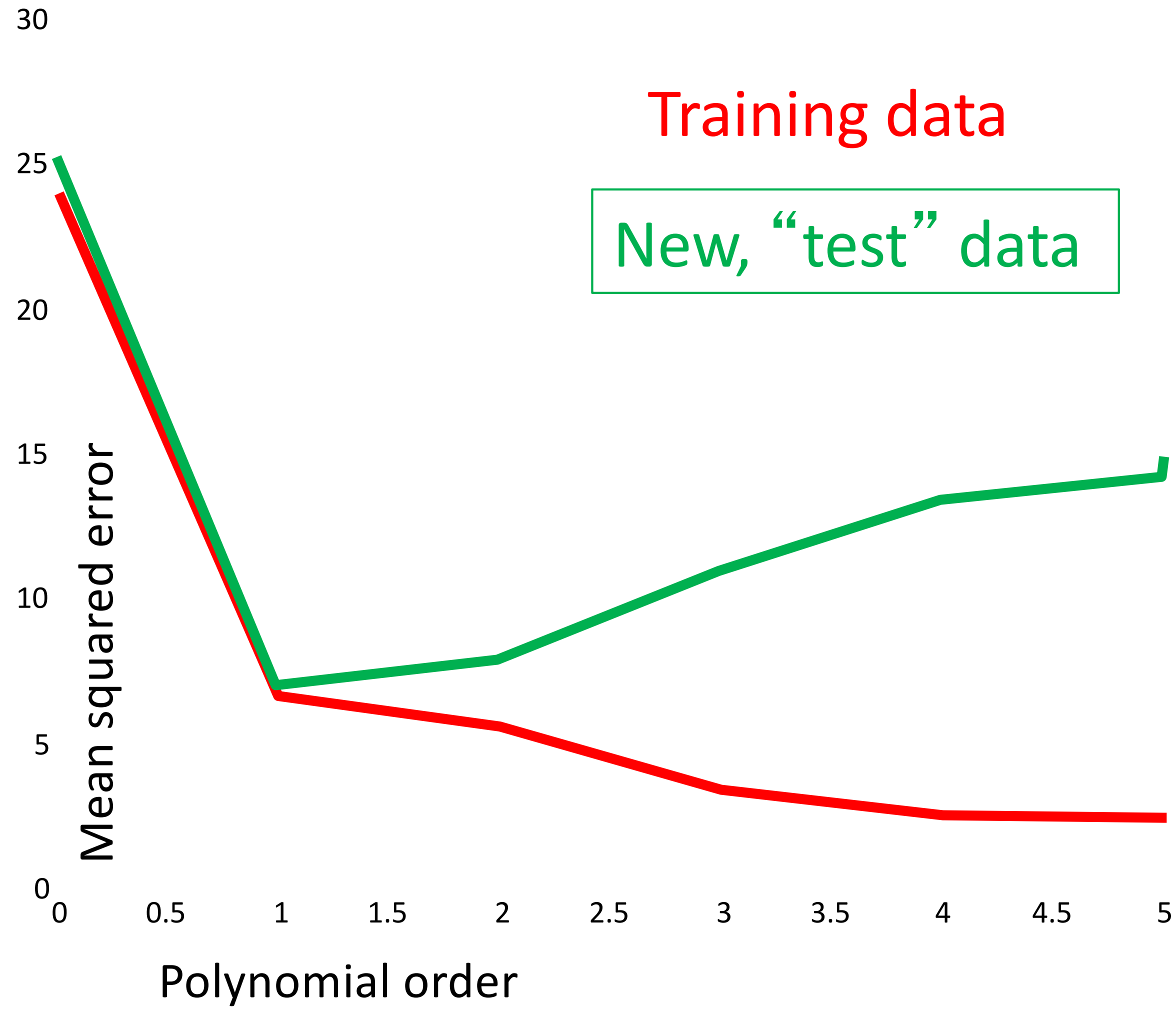
What about new data?

0th to 1st order

– Error decreases (Underfitting)

Higher order

– Error increases (Overfitting)



Inductive bias

The assumptions needed to predict examples we haven't seen

Some inductive bias is necessary for learning

Polynomial functions; smooth functions; etc

Generally prefer simpler models over more complex ones ("Occam's razor")