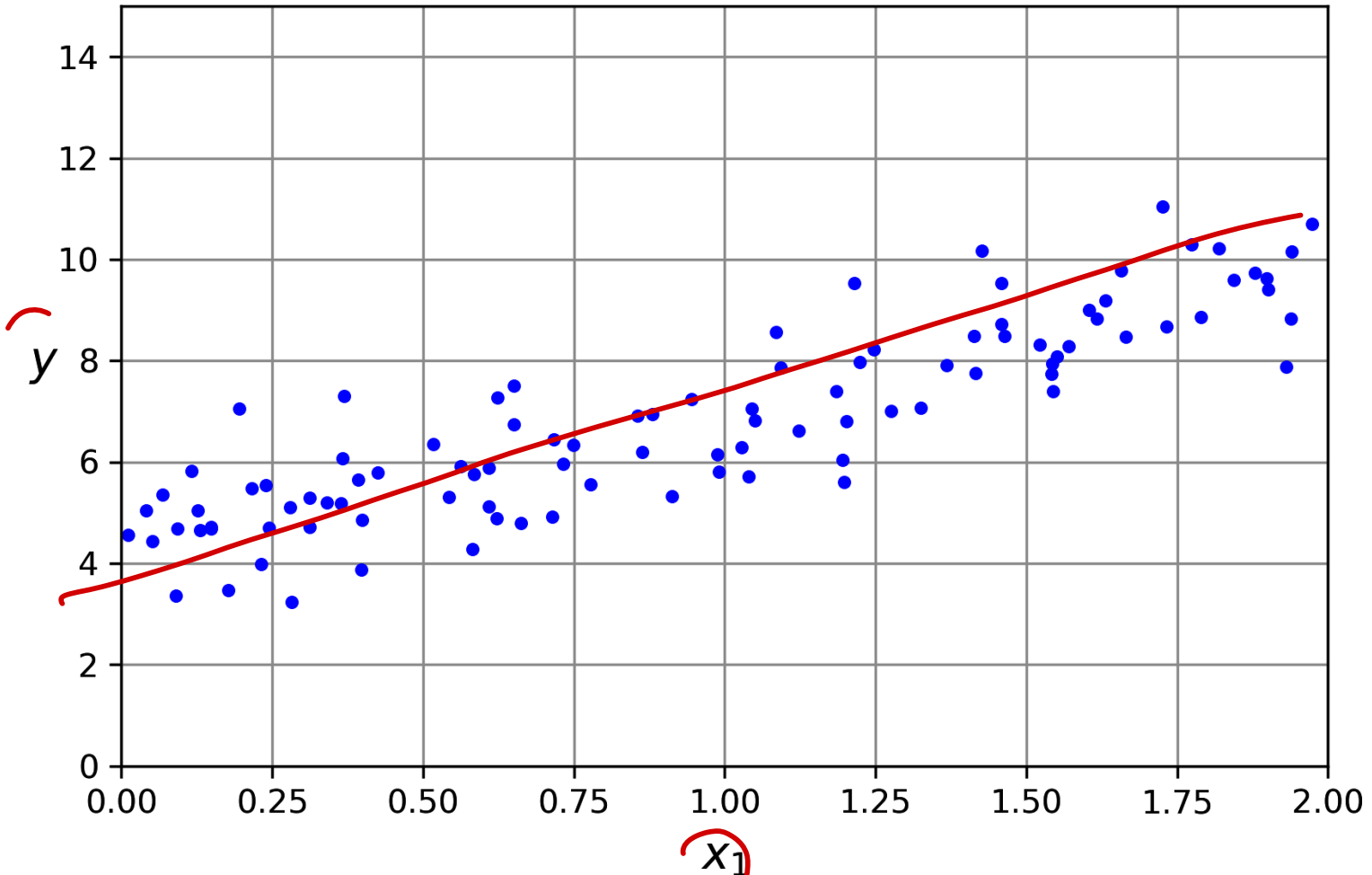
Linear Regression

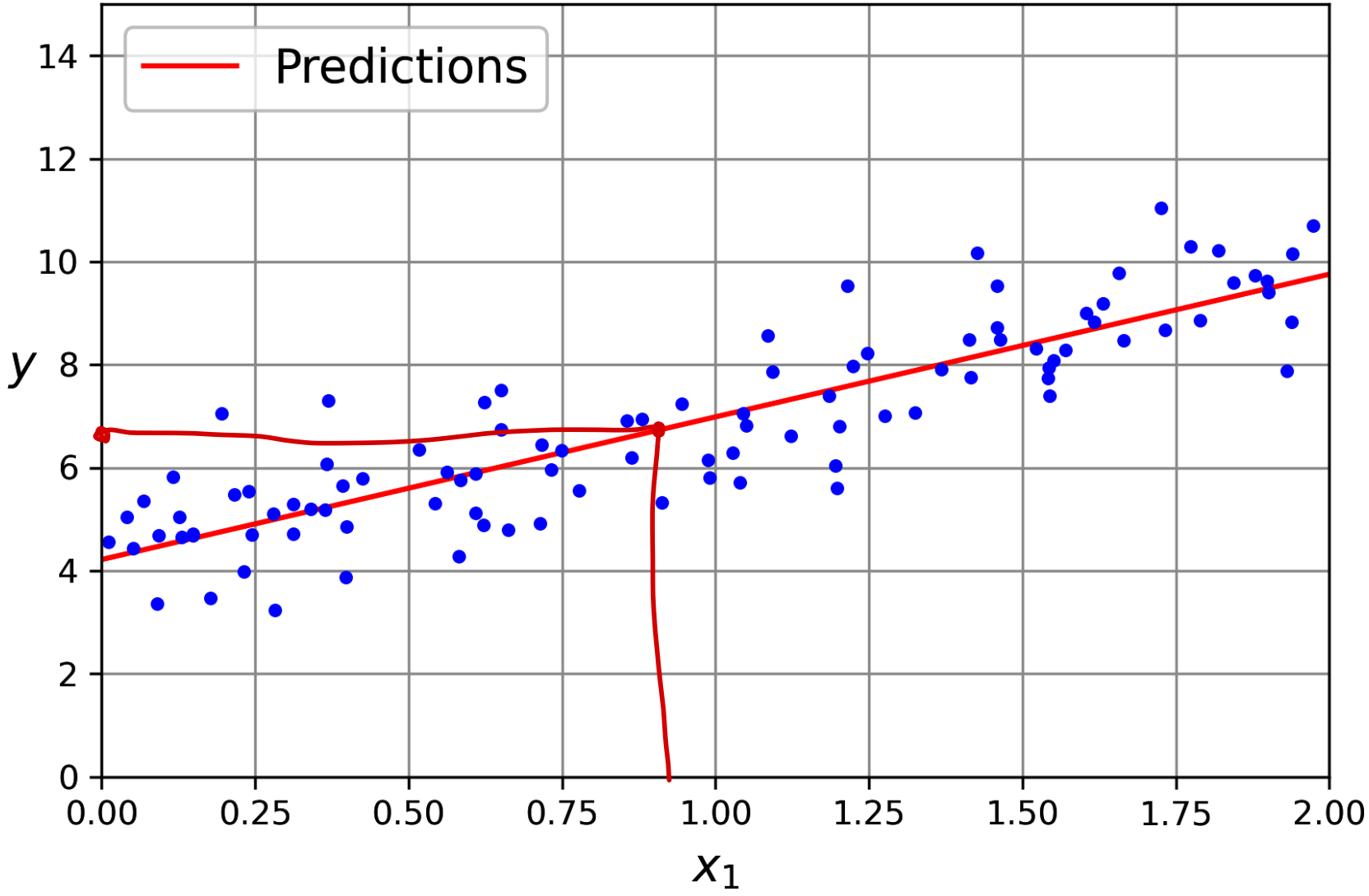
Lecture 3

Linear Regression

- Generating models which are a linear function of the input features.

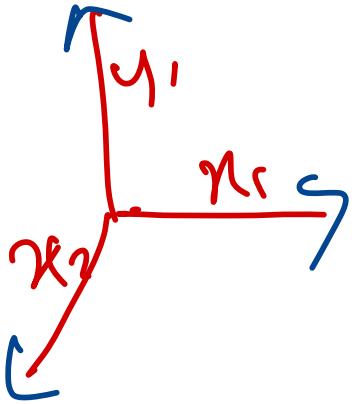


Linear Regression



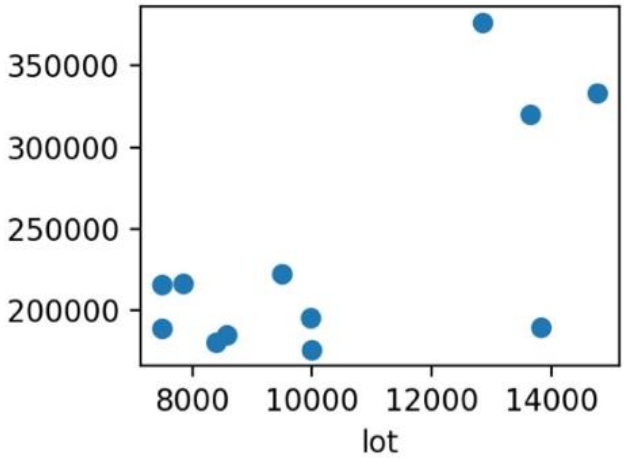
Hypothesis

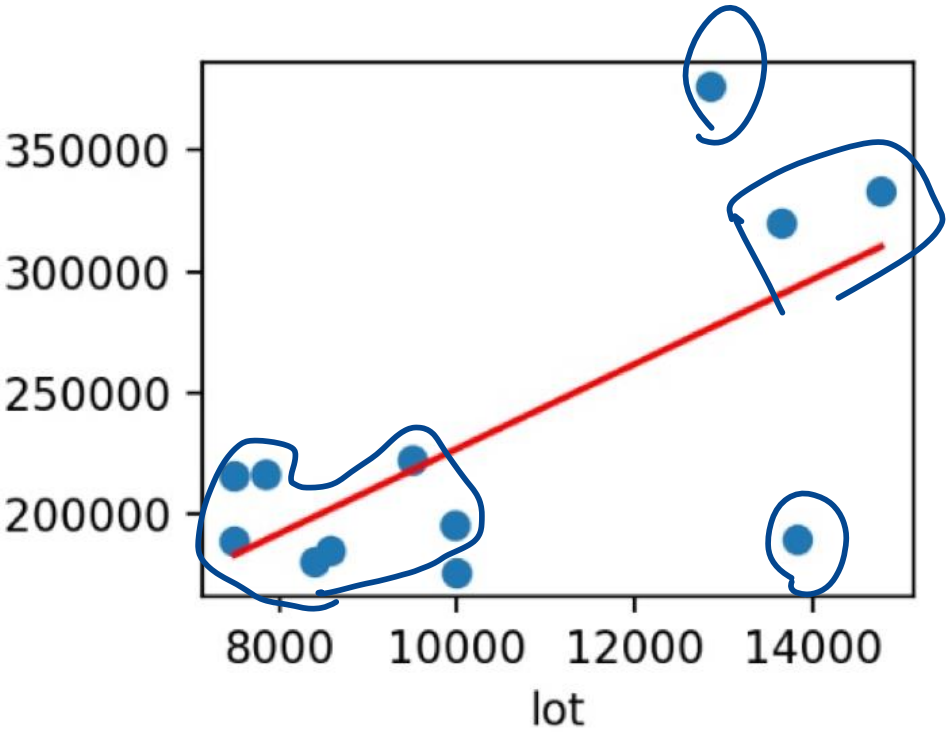
- A **hypothesis** or a prediction function is function $h : \mathcal{X} \to \mathcal{Y}$
  - $\mathcal{X}$ is an image, and $\mathcal{Y}$ contains "cat" or "not."
  - $\mathcal{X}$ is a text snippet, and $\mathcal{Y}$ contains "hate speech" or "not."
  - $\mathcal{X}$ is house data, and $\mathcal{Y}$ could be the price.
- A **training set** is a set of pairs $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$ s.t. $x^{(i)} \in \mathcal{X}$ and $y^{(i)} \in \mathcal{Y}$ for $i = 1, \ldots, n$.
- Given a training set our goal is to produce a *good* prediction function h
  - Defining "good" will take us a bit. It's a modeling question!
  - We will want to use h on *new* data not in the training set.



- If $\mathcal{Y}$ is continuous, then called a *regression problem*.
- If $\mathcal{Y}$ is discrete, then called a *classification problem*.

Housing Data





	SalePrice	Lot.Area
4	189900	13830
5	195500	9978
9	189000	7500
10	175900	10000
12	180400	8402
22	216000	7500
36	376162	12858
47	320000	13650
55	216500	7851
56	185088	8577
58	222500	9505
59	333168	14774

Hypothesis + More Features

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

Parameters.

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix} \quad x \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}$$

Living area ($feet^2$)	#bedrooms	Price (1000$s)	
2104	3	400	1
1600	3	330	2
2400	3	369	3

x_1 x_2 y^i

Two features – size, no of bedrooms …
Target variable is price.

$$h(x) = \sum_{j=0}^{n} \theta_j x_j$$

$h \Rightarrow$ hypothesis
$\theta \Rightarrow$ parameters
$n \Rightarrow$ no. of features.
$x \Rightarrow$ feature / input
$m \Rightarrow$ no. of training ex
$y \Rightarrow$ output / target var.
(x, y) training example
(x^i, y^i) example no. i^{th} example

$$\begin{cases} 400 = \theta_0 + 2104\theta_1 + 3\theta_2 & 1 \\ 330 = \theta_0 + 1600\theta_1 + 3\theta_2 & 2 \\ 369 = \theta_0 + 2400\theta_1 + 3\theta_2 & 3 \end{cases}$$

Cost Function

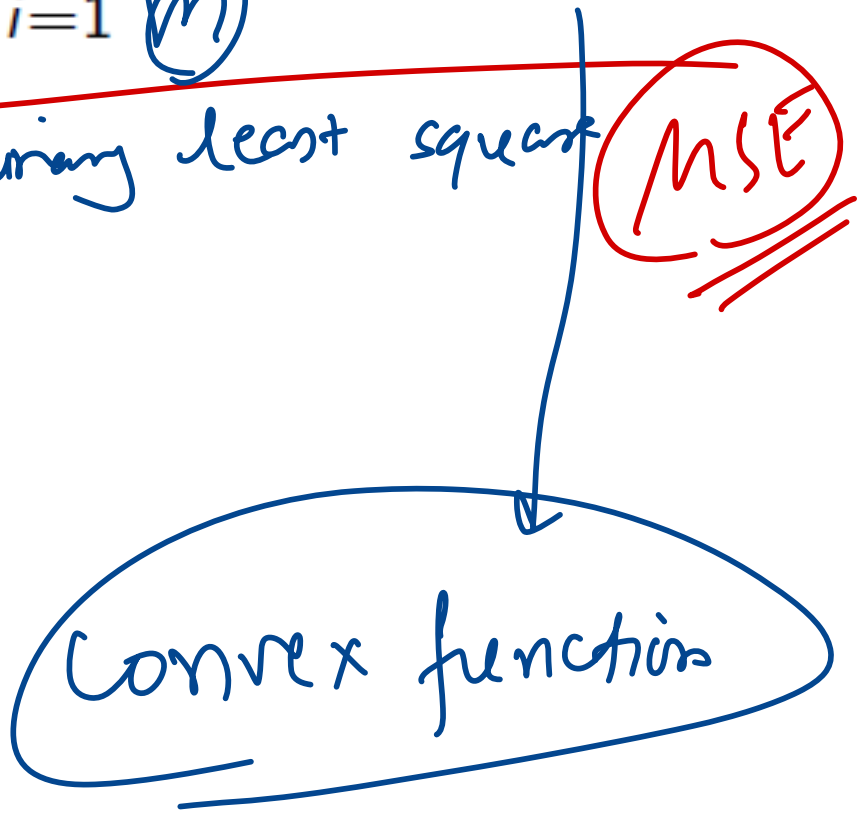
$$J(\theta) = \frac{1}{2}\sum_{i=1}^{n}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

$h_\theta(x) \Rightarrow y$

Find θ to min $J(\theta)$

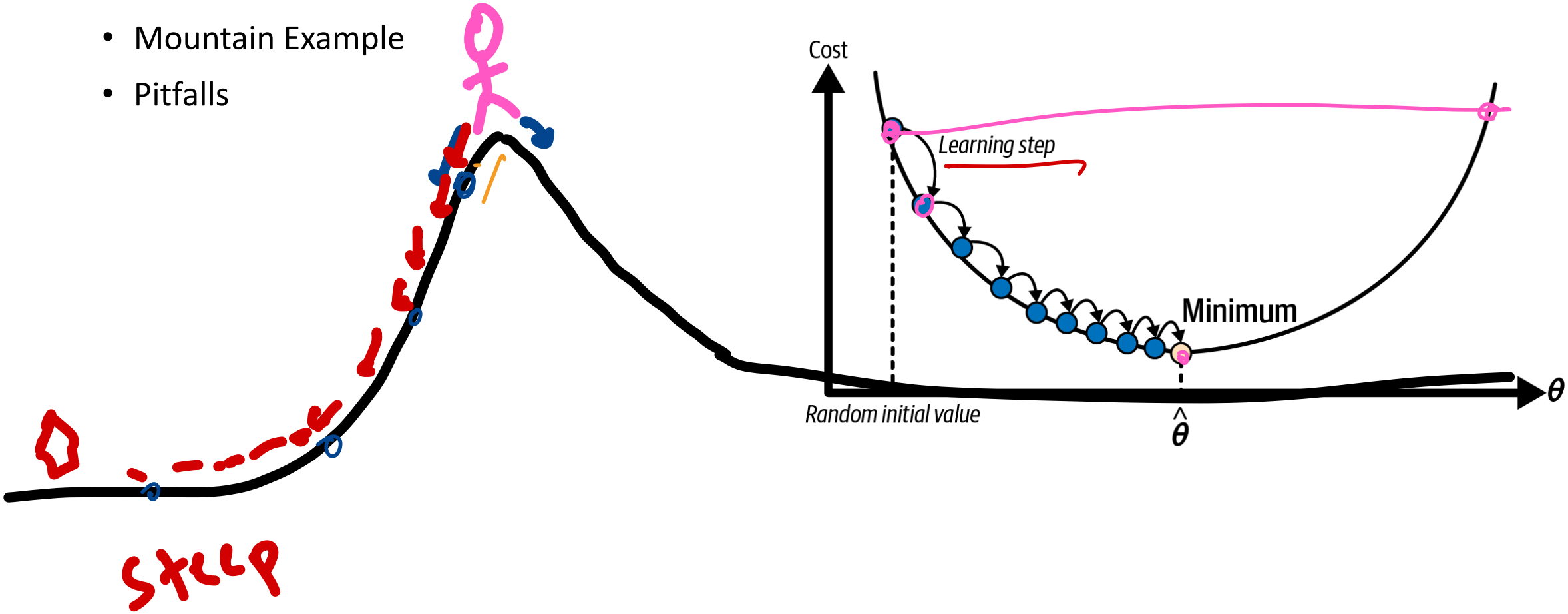
ordinary least square

MSE



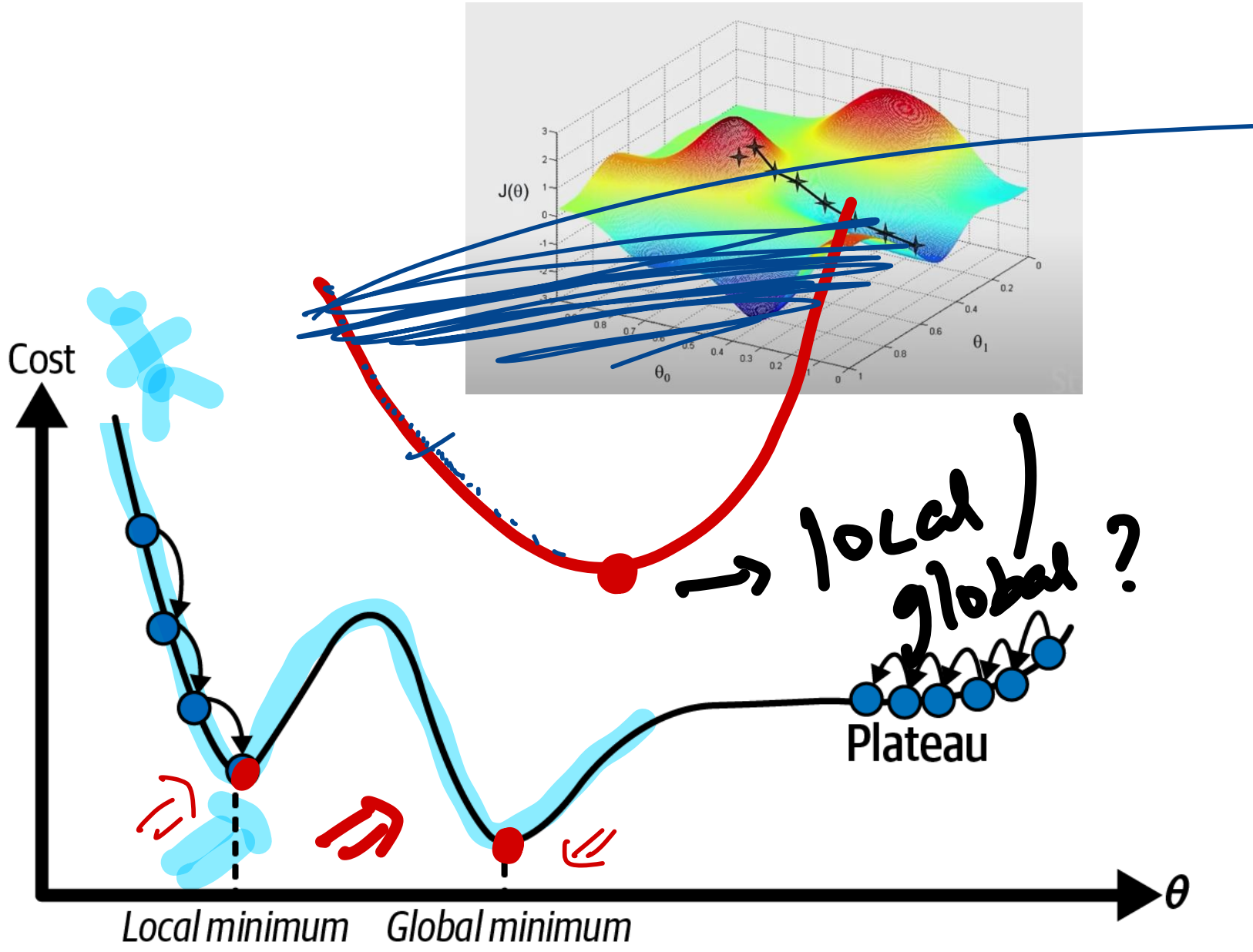
Least Squares Optimization Problem

- *Gradient descent* is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of gradient descent is to tweak parameters iteratively in order to minimize a cost function.
- Mountain Example
- Pitfalls



Pitfalls

- Local optima for irregular terrain
- The MSE cost function is convex which means that the is no local minima
- Convex function any two points on the curve, the line segment joining them is never below the curve.



Andrew Ng CS229

Iterations for finding Θ

Start with some value of θ $\vec{\theta} = 0$

Find $\theta_0, \theta_1, \theta_2 \ldots$ s.t minimise $J(\theta)$

$\theta_j^{t+1} = \theta_j^t - \alpha \frac{\partial}{\partial \theta} (J(\theta)^t)$

$\alpha \Rightarrow$ learning rate.

j → iterator

α → learning rate

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{\partial}{\partial \theta_j} \frac{1}{2} (h_\theta(x) - y)^2$$

$$= 2 \times \frac{1}{2} (h_\theta(x) - y) \frac{\partial}{\partial \theta_j} (h_\theta(x) - y)$$

$\Rightarrow$

0 - 100

0 - 50

20 - 30

0 — 100000

-1 - 1

v.large x_1 + v.small

$$(h_\theta(x) - y)\left[\frac{\partial}{\partial \theta_j}\left(\theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n - y\right)\right]$$

All terms are 0 except for terms corresponding to j

$$(h_\theta(x) - y) \cdot x_j$$

θ_0 θ_1 θ_2 θ_j

$$\theta_j^{t+1} = \theta_j^t - \alpha\left(\frac{\partial}{\partial \theta_j} J(\theta)\right)$$

$$\theta_j^{t+1} = \theta_j^t + \alpha\left(y^{(i)} - h_\theta(x^{(i)})\, x_j^{(i)}\right) \Rightarrow$$

θ_0 θ_1 θ_2 θ_n

Repeat until convergence for $j = (0, 1, \cdots n)$

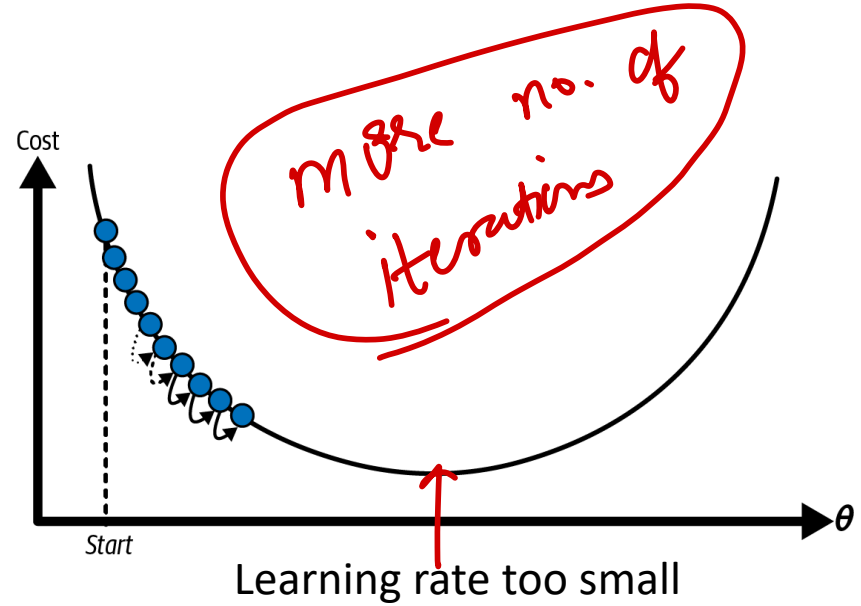
Iterations for finding Θ

Prescribe no. of iterations $\Rightarrow$ 1000

$J < 10^{-6}$

Learning Rate

- Learning rate hyperparameter

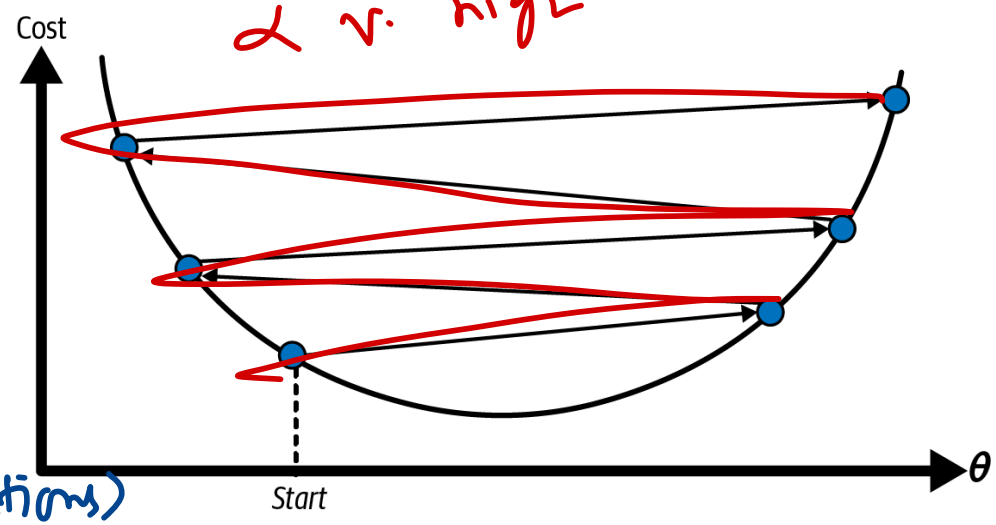


α value ?

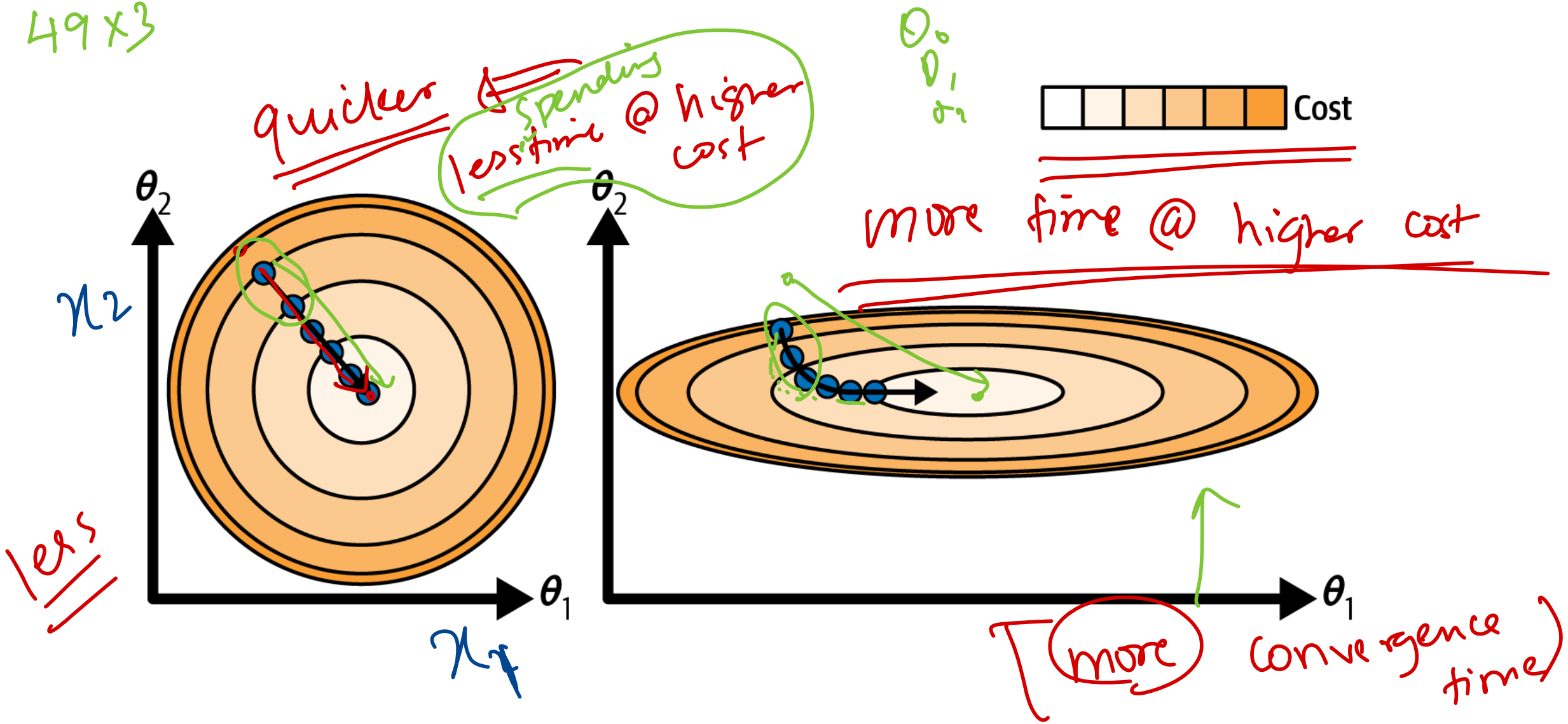
Convergence criteria

1) Fix no. of iterations

2) threshold on the $J(\theta) \leq 10^{-4}$

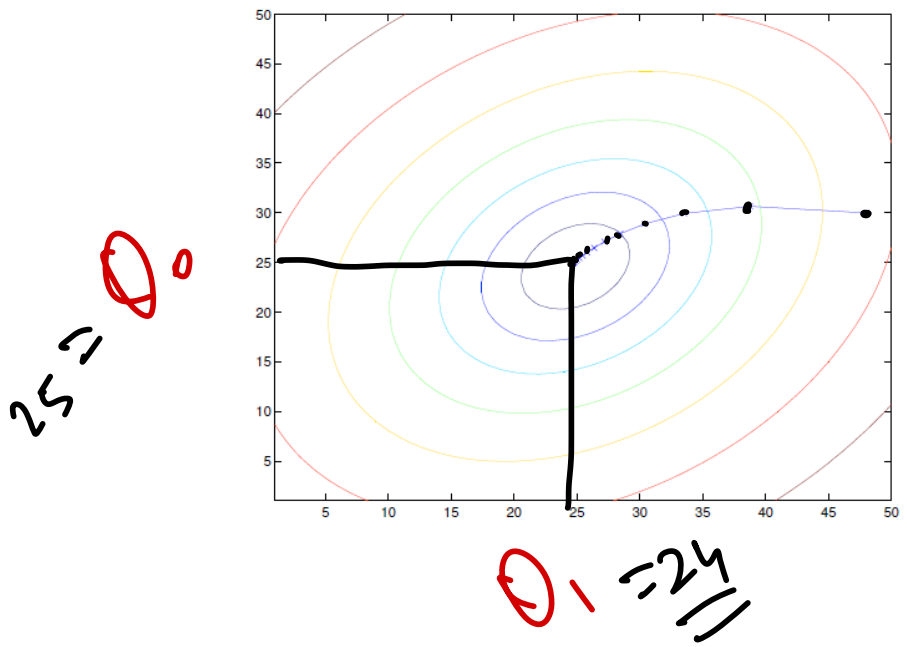


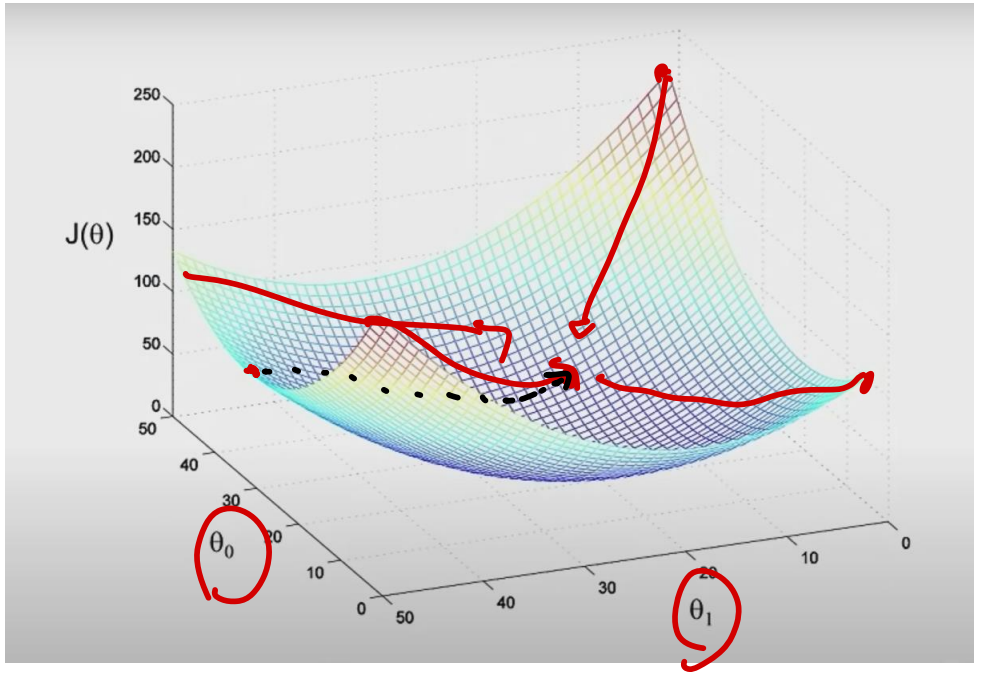
Feature Scaling for Gradient Descent



Cost Function

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{n}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$





Convex Function
Slices of the convex function
Contours of the quadratic function.

Solution

$\theta_0 = 89.60, \theta_1 = 0.1392, \theta_2 = -8.738$

- 49 – housing price data
- Parameters are updated according to the gradient of the error with respect to that single training example only

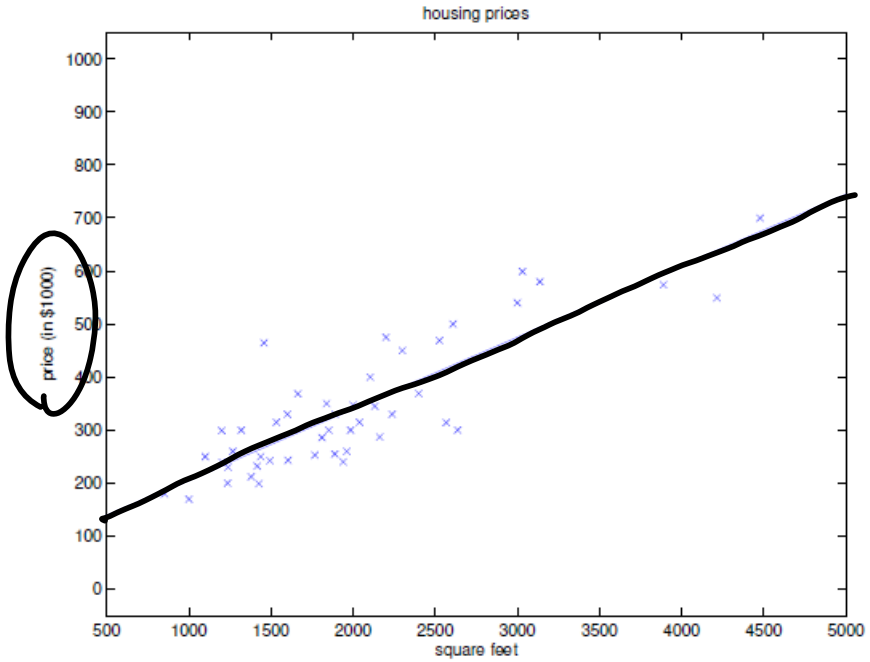
Loop {

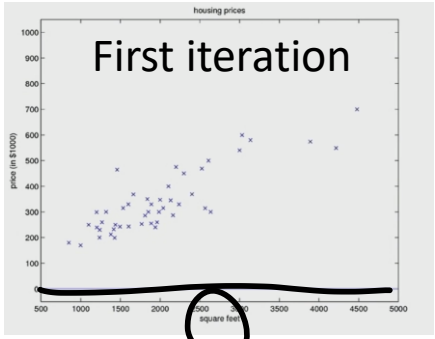
for $i = 1$ to n, {

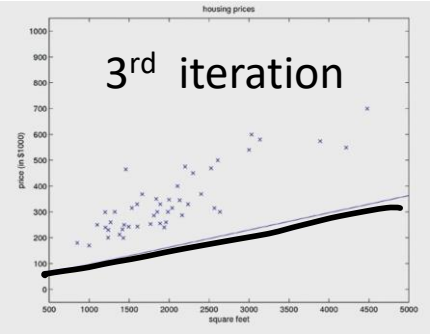
$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}, \quad \text{(for every } j\text{)}$$

}

}







Batch Gradient Descent vs. Stochastic GD.

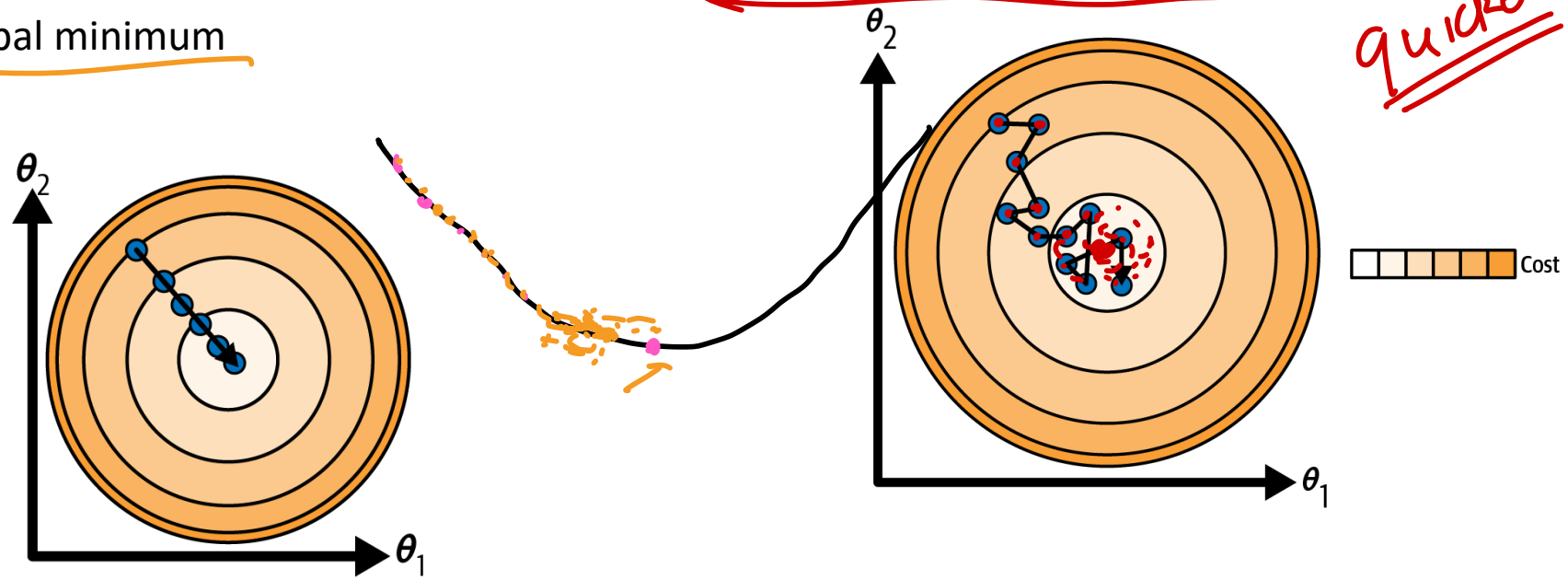
- The main problem with batch gradient descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large.

Slows down.

- *Stochastic gradient descent* picks a random instance in the training set at every step and computes the gradients based only on that single instance.
- Stochastic gradient descent will not find out the final minimum – will bounce around the minimum
- Will oscillate near the global minimum
- ~~A~~ Depends on learning rare

Reduce

Quicker



Stochastic Gradient Descent with Dynamic Learning Schedule

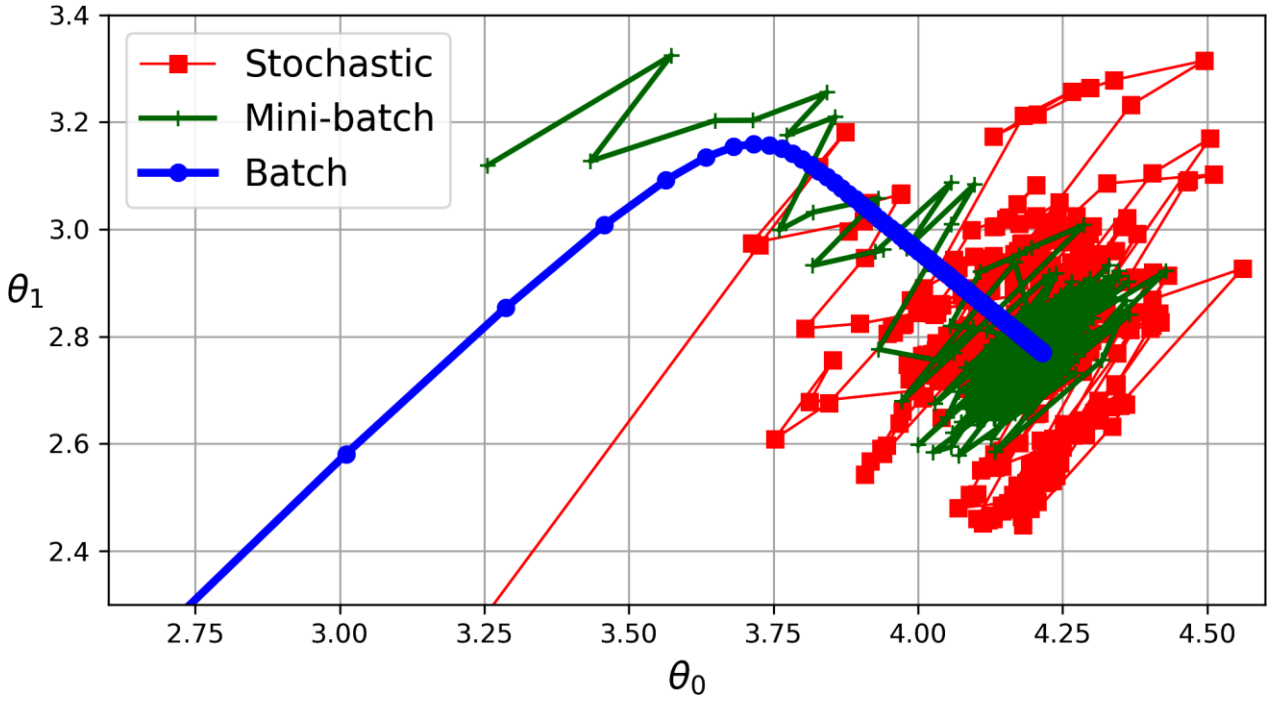
```
n_epochs = 50
t0, t1 = 5, 50  # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

np.random.seed(42)
theta = np.random.randn(2, 1)  # random initialization

for epoch in range(n_epochs):
    for iteration in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index : random_index + 1]
        yi = y[random_index : random_index + 1]
        gradients = 2 * xi.T @ (xi @ theta - yi)  # for SGD, do not divide by m
        eta = learning_schedule(epoch * m + iteration)
        theta = theta - eta * gradients
```

Linear

dynamic α

Mini Batch GD

- Computes gradients on small random sets of instances called mini-batches.
- A performance boost is obtained in terms of hardware optimization of matrix operations, esp. for GPUs.
- Algorithm is less errratice as compared to stochastic GD.
- Batch GD finds the exact solution – takes more time though.



Normal Equation Derivation

$\theta_0 \; \theta_1 \; \theta_2$

$$\hat{\theta} = (X^T X)^{-1} X^T y \Leftarrow$$

$$\theta_0 = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}$$

$n+1 \times 1$

X

$m \times (n+1)$

$$\begin{bmatrix} \cdots & x^{(1)T} & \cdots \\ \cdots & x^{(2)T} & \cdots \\ & \vdots & \\ \cdots & x^{(m)T} & \cdots \end{bmatrix}$$

↓ training ex.

$$x^i = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

→ features.

$X \; \theta \Rightarrow$ **$m \times 1$**

$$X\theta = \begin{bmatrix} x^{(1)}\theta \\ \vdots \\ x^{(m)}\theta \end{bmatrix} = \begin{bmatrix} h_\theta(x^{(1)}) \\ \vdots \\ h_\theta(x^{(m)}) \end{bmatrix}$$

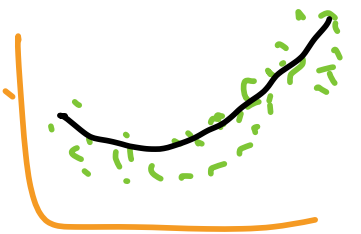
$$\vec{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}$$

$$(X\theta - y) = \begin{pmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ \vdots \\ h_\theta(x^{(m)}) - y^{(m)} \end{pmatrix}$$

$$\nabla_\theta J(\theta) = \begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \end{bmatrix} \; ! \; \frac{\partial J}{\partial \theta_n}$$

$\theta \in \mathbb{R}^{n+1}$

$(n+1 \times 1)$



$$\frac{1}{2}(X\theta - y)^T (X\theta - y) \Longrightarrow J(\theta)$$

$$z^T z = \sum_i z_i^2$$

$$\frac{1}{2}\sum \left(h_\theta(x^{(i)} - y^{(i)} \right)^2$$

Normal Equation

- Gives the result directly for the MSE equation.

gradient

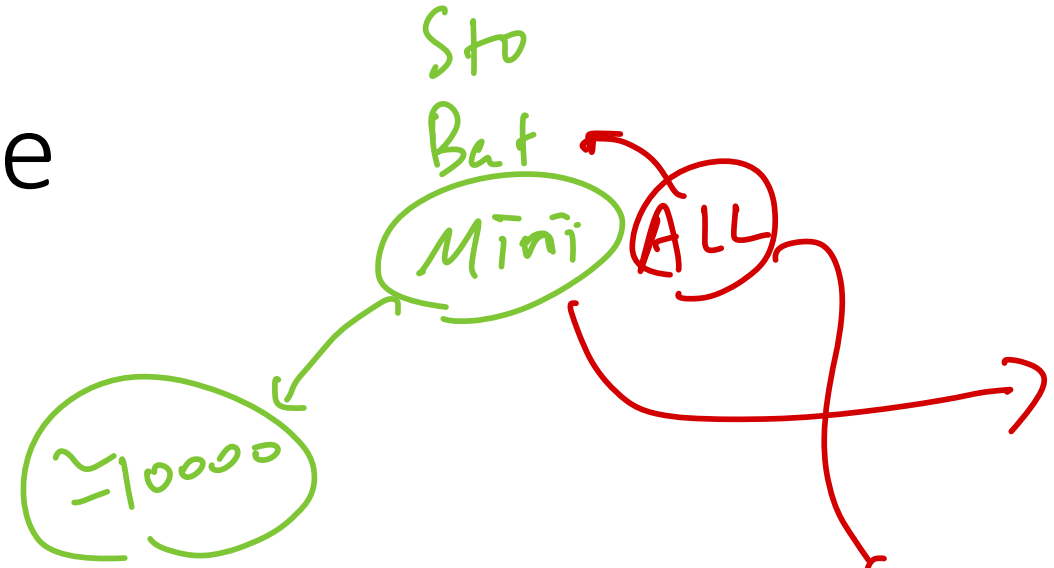
$$
\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \frac{1}{2}(X\theta - \vec{y})^T(X\theta - \vec{y}) \\
&= \frac{1}{2}\nabla_\theta \left((X\theta)^T X\theta - (X\theta)^T\vec{y} - \vec{y}^T(X\theta) + \vec{y}^T\vec{y}\right) \\
&= \frac{1}{2}\nabla_\theta \left(\theta^T(X^TX)\theta - \vec{y}^T(X\theta) - \vec{y}^T(X\theta)\right) \\
&= \frac{1}{2}\nabla_\theta \left(\theta^T(X^TX)\theta - 2(X^T\vec{y})^T\theta\right) \\
&= \frac{1}{2}\left(2X^TX\theta - 2X^T\vec{y}\right) \\
&= X^TX\theta - X^T\vec{y}
\end{aligned}
$$

Thus, the value of θ that minimizes $J(\theta)$ is given in closed form by the equation

$$\theta = (X^TX)^{-1}X^T\vec{y}$$

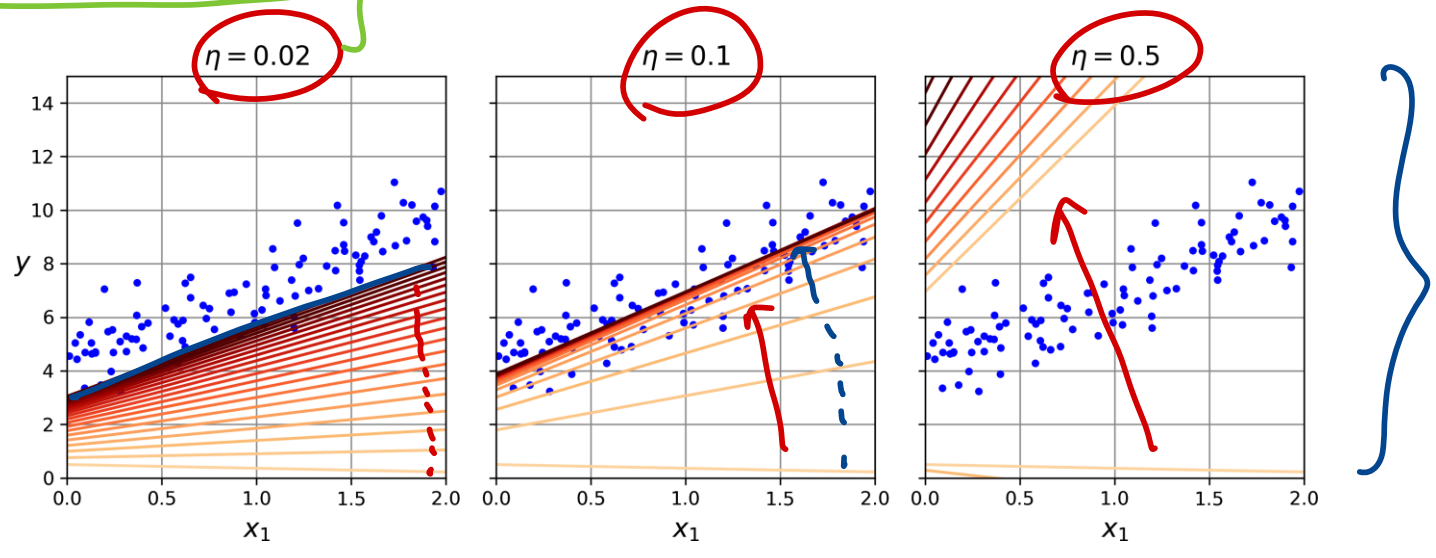
Large dataset

Normal Equation code



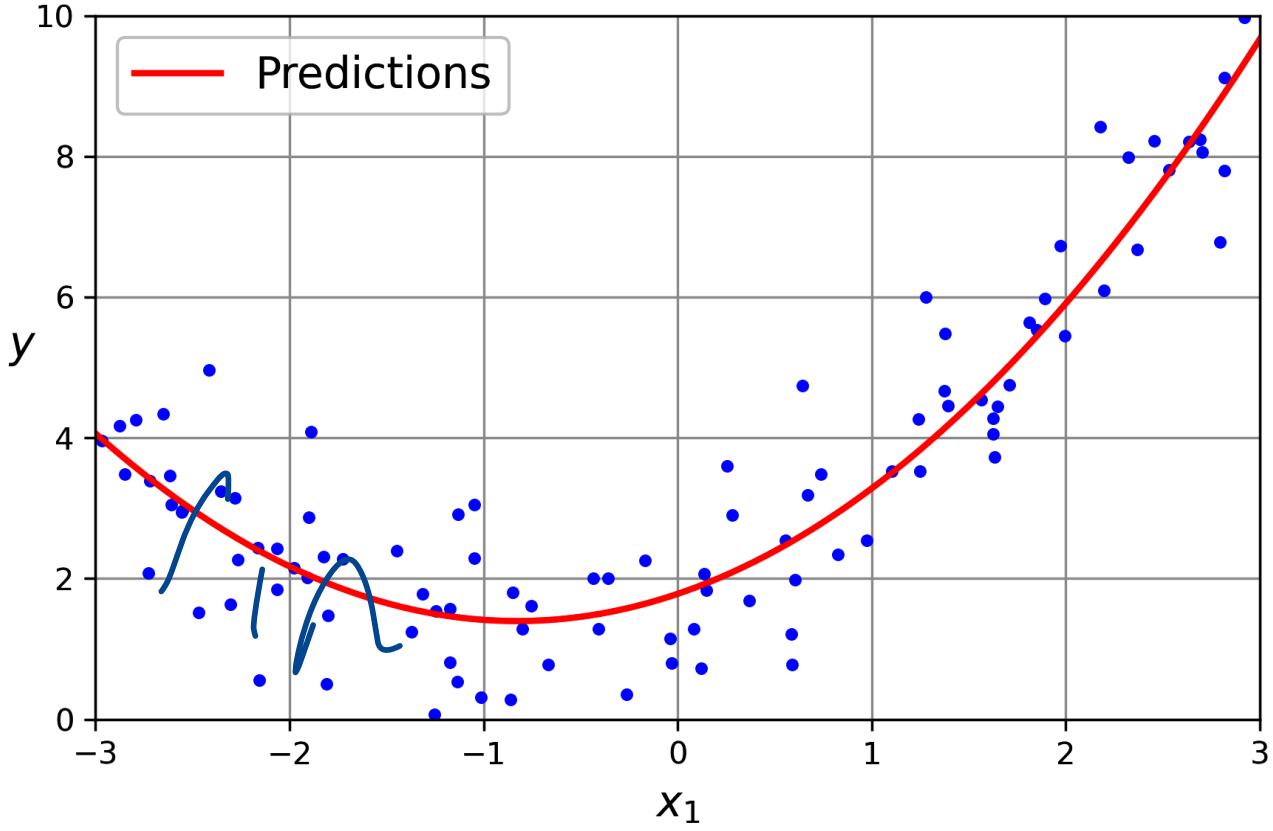

```
eta = 0.1  # learning rate
n_epochs = 1000
m = len(X_b)  # number of instances

np.random.seed(42)
theta = np.random.randn(2, 1)  # randomly initialized model parameters

for epoch in range(n_epochs):
    gradients = 2 / m * X_b.T @ (X_b @ theta - y)
    theta = theta - eta * gradients
```

```
>>> theta
array([[4.21509616],
       [2.77011339]])
```



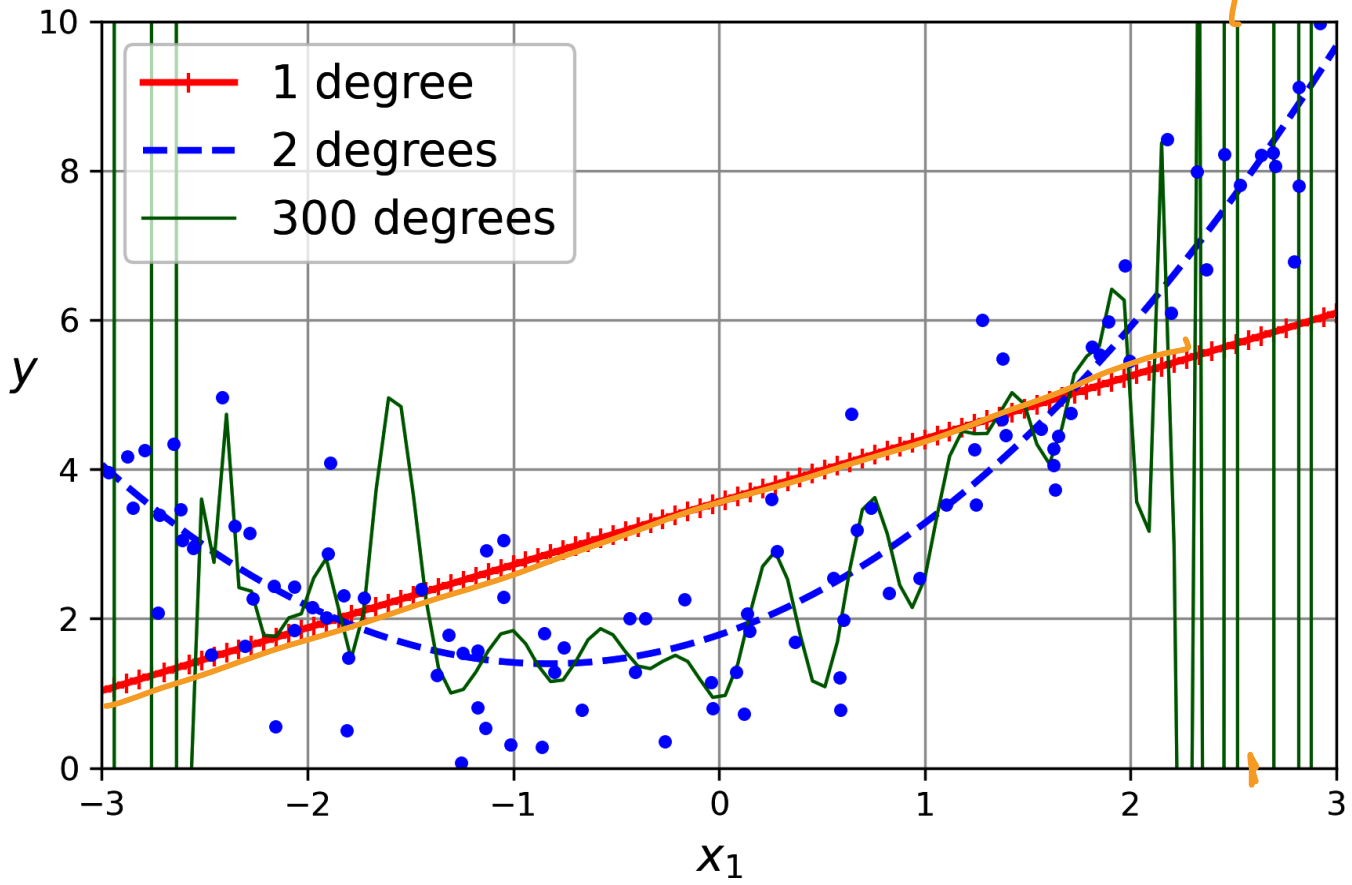
Polynomial Regression

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929,  0.56664654])


>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```



Not bad: the model estimates $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$ when in fact the original function was $y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Gaussian noise}$.

High Order Polynomial – Nice fit ?



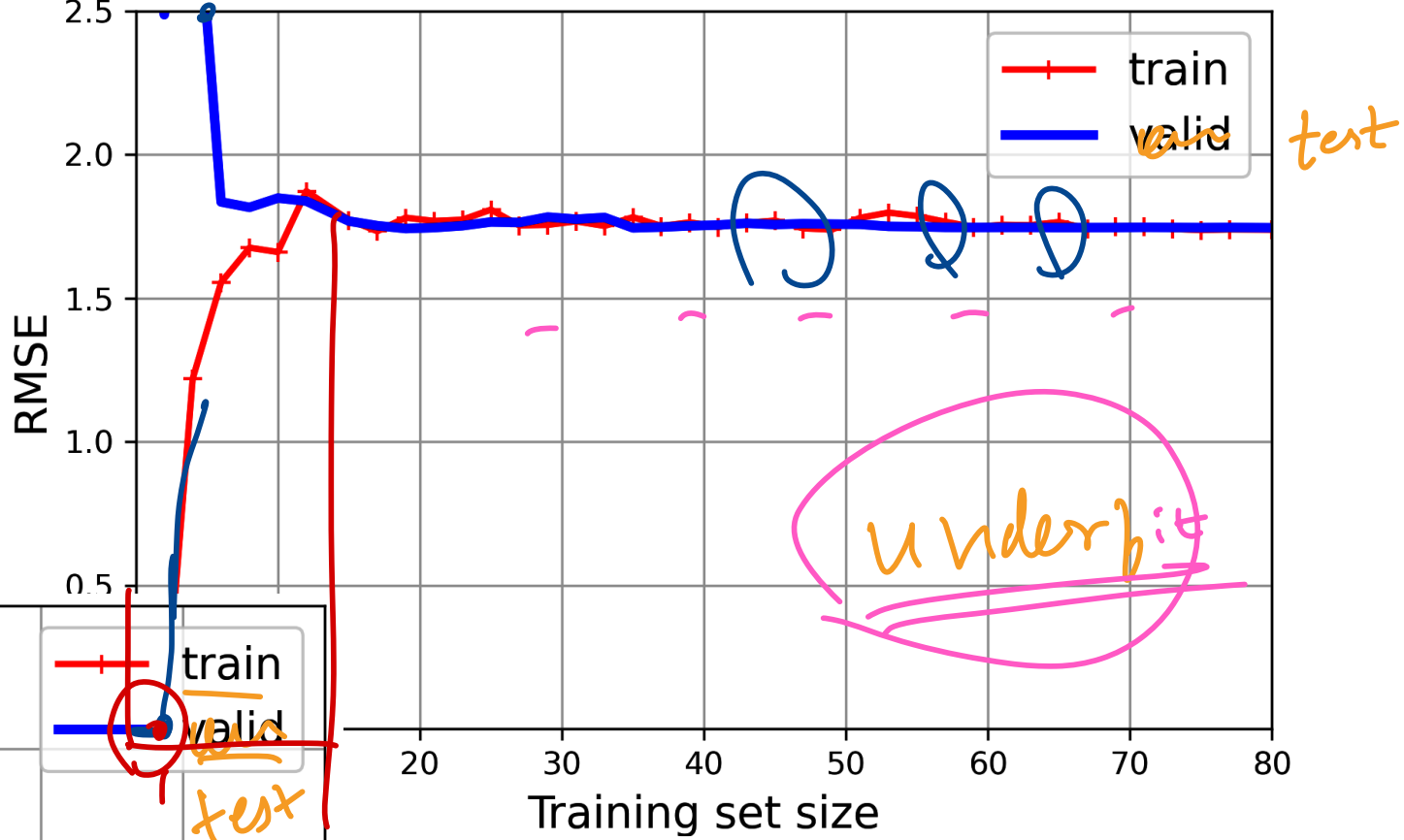
Learning Curves

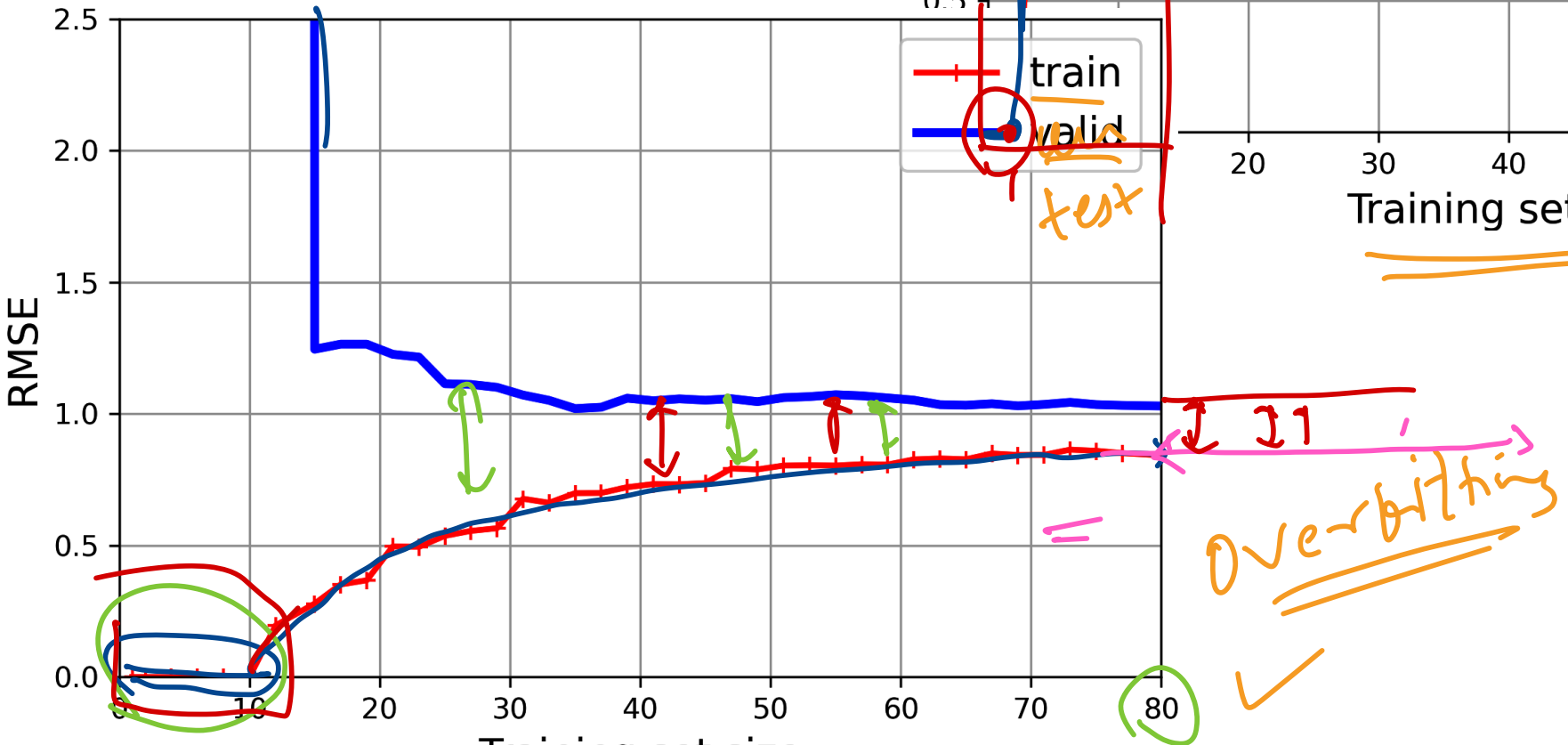
3rd ed.

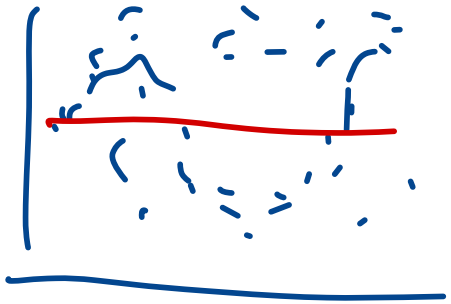
Hands on Machine with Scikit

Pg 153 154

- Under fitting
- Overfitting





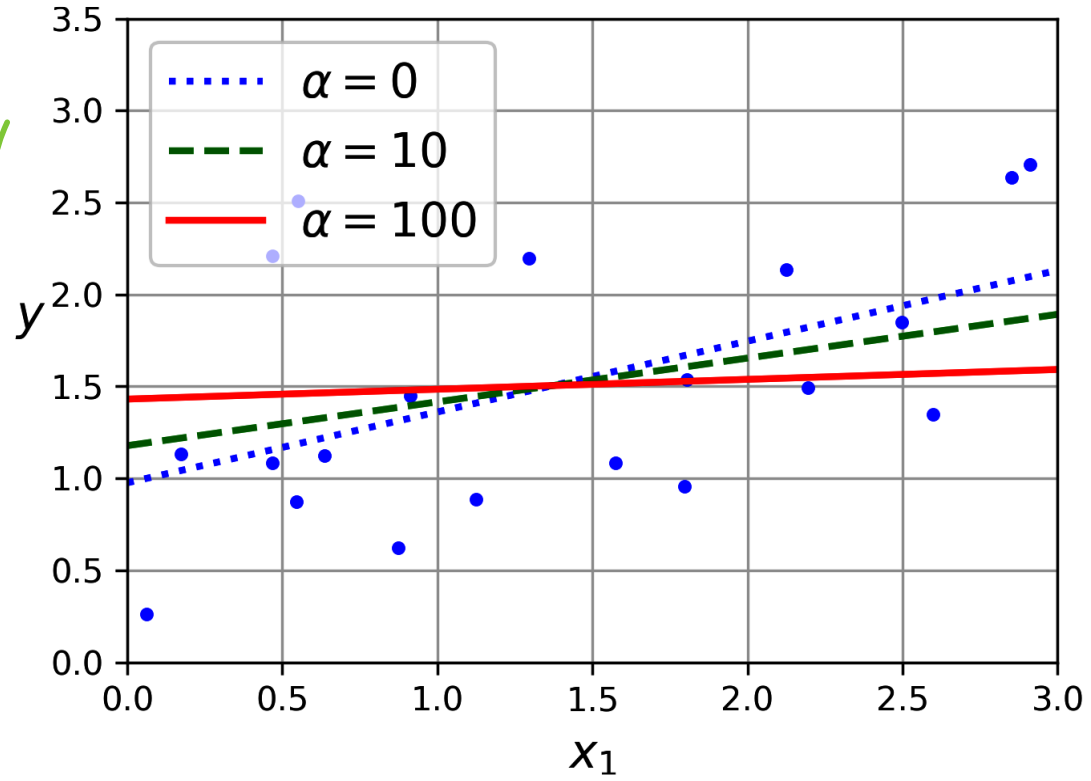


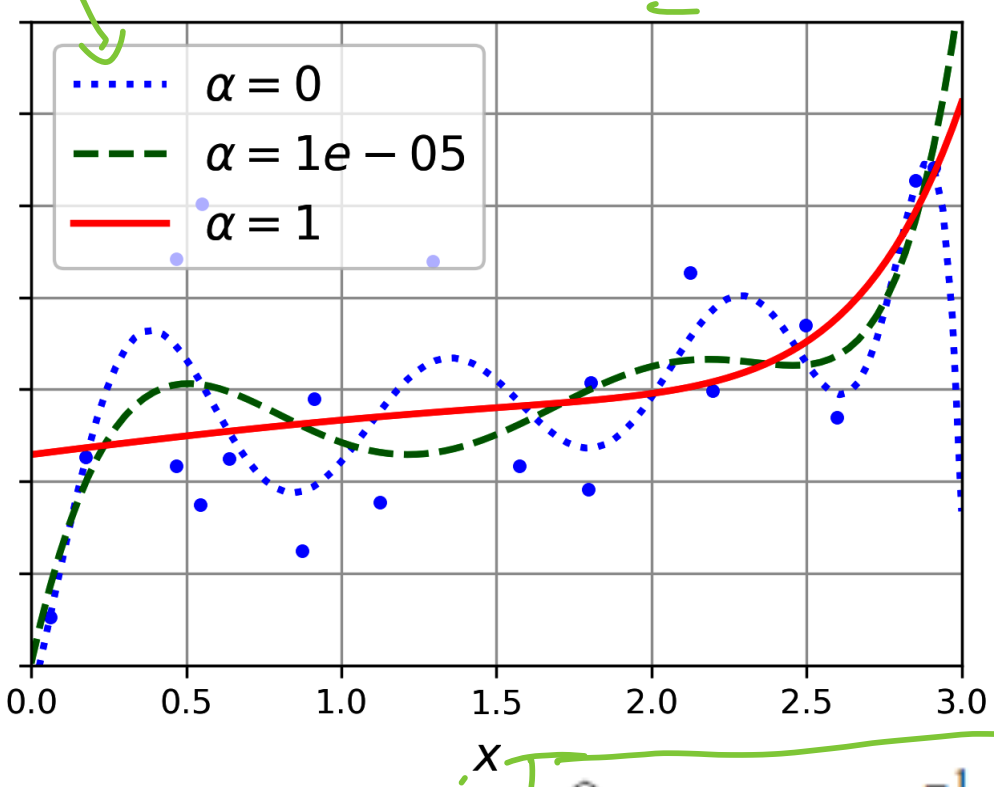
Regularized Linear Models – tackles overfitting

- Ridge regression

Linear and polynomial modes.

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \frac{\alpha}{m}\Sigma_{i=1}^{n}\theta_i^2$$



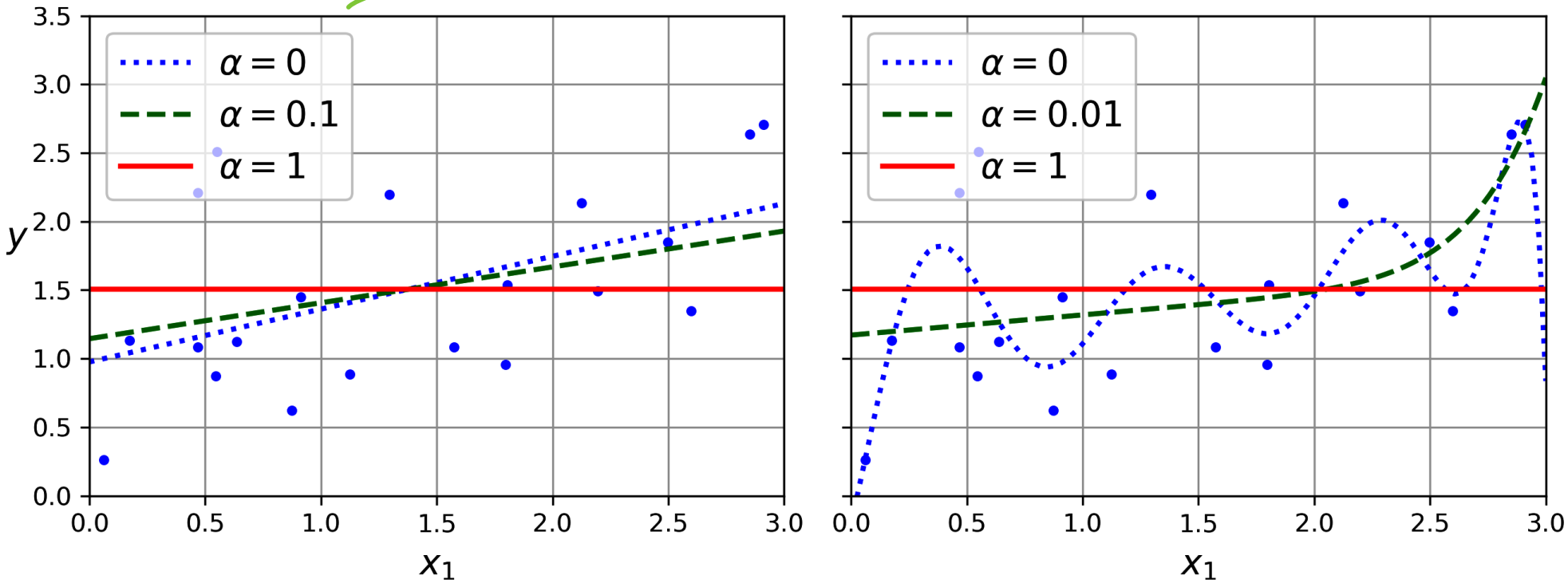


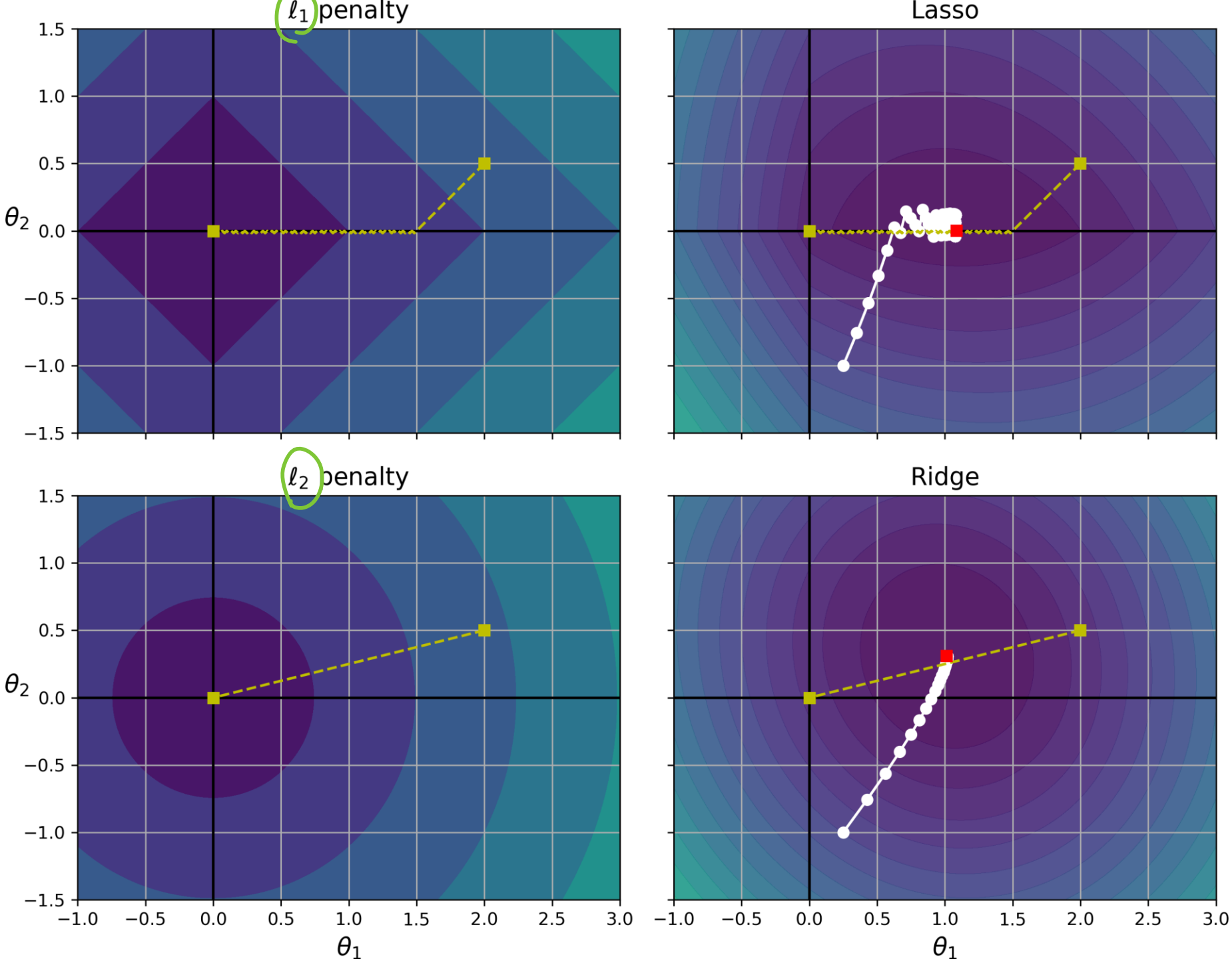
- Closed form solution for Ridge Regression

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^{\intercal}\mathbf{X} + \alpha\mathbf{A})^{-1}\ \mathbf{X}^{\intercal}\ \mathbf{y}$$

Lasso Regression

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + 2\alpha\Sigma_{i=1}^{n}|\theta_i|$$




ℓ_1 penalty
Lasso
ℓ_2 penalty
Ridge
θ_1
θ_2
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0

Elastic Net Regression

- A middle ground between ridge and lasso regression
- Weighted sum of both Ridge and Lasso regression - mix ratio *r*
- *r=0* … Ridge regression
- *r=1* … Lasso regression

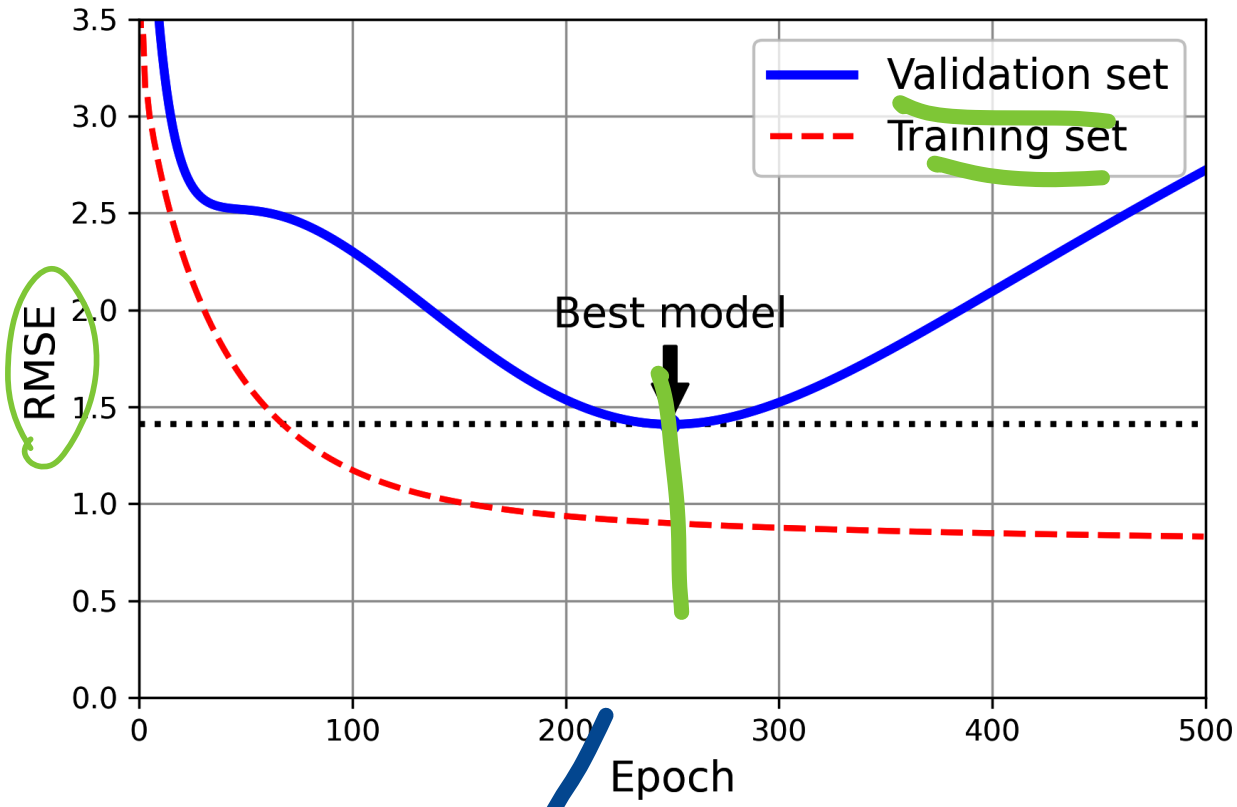
$$J(\mathbf{\theta}) = \mathrm{MSE}(\mathbf{\theta}) + r\left(2\alpha\Sigma_{i=1}^{n}\left|\theta_i\right|\right) + \left(1-r\right)\left(\frac{\alpha}{m}\Sigma_{i=1}^{n}\theta_i^2\right)$$

Which Regression

- Plain, Ridge, Lasso, Elastic Net ?
- Preferable to have some regularization
- Ridge is a good default
- If you feel only a few features are useful – prefer Lasso or Elsatic Net as they reduce the weights of useless features to 0.
- Elastic net is preferred as compared to Lasso …

Early stopping

- Another different regulsarization:
- Stop training when the validation error reaches a minimum – Early stopping
- After min validation error – overfitting has started.



Some Code …

```
from copy import deepcopy
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler

X_train, y_train, X_valid, y_valid = [...]  # split the quadratic dataset

preprocessing = make_pipeline(PolynomialFeatures(degree=90, include_bias=False),
                              StandardScaler())
X_train_prep = preprocessing.fit_transform(X_train)
X_valid_prep = preprocessing.transform(X_valid)
sgd_reg = SGDRegressor(penalty=None, eta0=0.002, random_state=42)
n_epochs = 500
best_valid_rmse = float('inf')

for epoch in range(n_epochs):
    sgd_reg.partial_fit(X_train_prep, y_train)
    y_valid_predict = sgd_reg.predict(X_valid_prep)
    val_error = mean_squared_error(y_valid, y_valid_predict, squared=False)
    if val_error < best_valid_rmse:
        best_valid_rmse = val_error
        best_model = deepcopy(sgd_reg)
```