Multiple features (variables)

Size in feet² x_1	Number of bedrooms x_2	Number of floors x_3	Age of home in years x_4	Price ($) in $1000's
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
...	...	...	...	...

$j = 1 \ldots 4$

$n = 4$

$i=2$

$x_j = j^{th}$ feature

n = number of features

$\vec{x}^{(i)}$ = features of i^{th} training example

$x_j^{(i)}$ = value of feature j in i^{th} training example

X superscript i, subscript j

Raw vector(List)

$\vec{x}^{(2)} = [1416 \;\; 3 \;\; 2 \;\; 40]$

$x_3^{(2)} = 2$

Model: **Single linear regression model:**

Previously: $f_{w,b}(x) = wx + b$

$$f_{w,b}(X) = w_1X_1 + w_2X_2 + w_3X_3 + w_4X_4 + b$$

example **Multiple linear regression model example**

$$f_{w,b}(X) = 0.1X_1 + 4X_2 + 10X_3 + -2X_4 + 80$$

size #bedrooms #floors years base price

$$f_{w,b}(x) = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$$

80 → Base price while assuming that the house has no size, no bedrooms, no floors, no age parameters.(Start price).

0.1 X1 → Means that the price will increase by (0.1*1000 = 100 $) for each additional square feet.

4 X2 → Means that the price will increase by (4*1000 = 4000 $) for each additional bedroom.

10 X3 → Means that the price will increase by (10*1000 = 10000 $) for each additional floor.

-2 X4 → Means that the price will decreases by (2*1000 = 2000 $) for each additional year of age.

$$f_{\vec{w},b}(\vec{x}) = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$$

$\vec{w} = [w_1 \ w_2 \ w_3 \ \ldots \ w_n]$

b is a number

parameters of the model

vector

$\vec{x} = [x_1 \ x_2 \ x_3 \ \ldots \ x_n]$

This arrow is optional

Note:
Raw vector = List

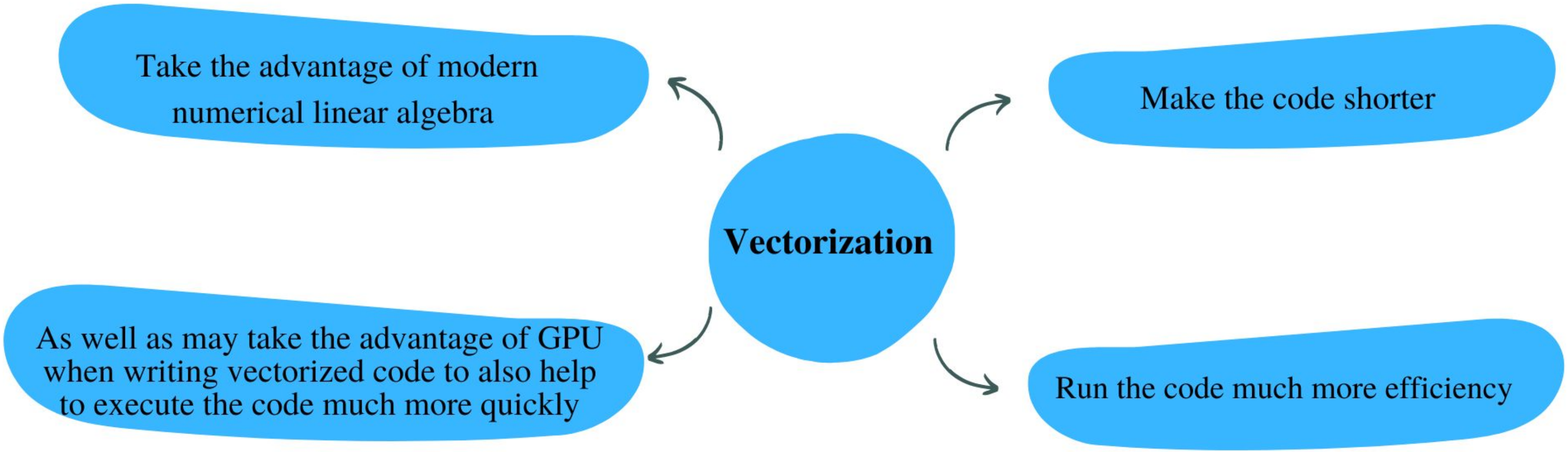
$$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b = w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + b$$

dot product

multiple linear regression

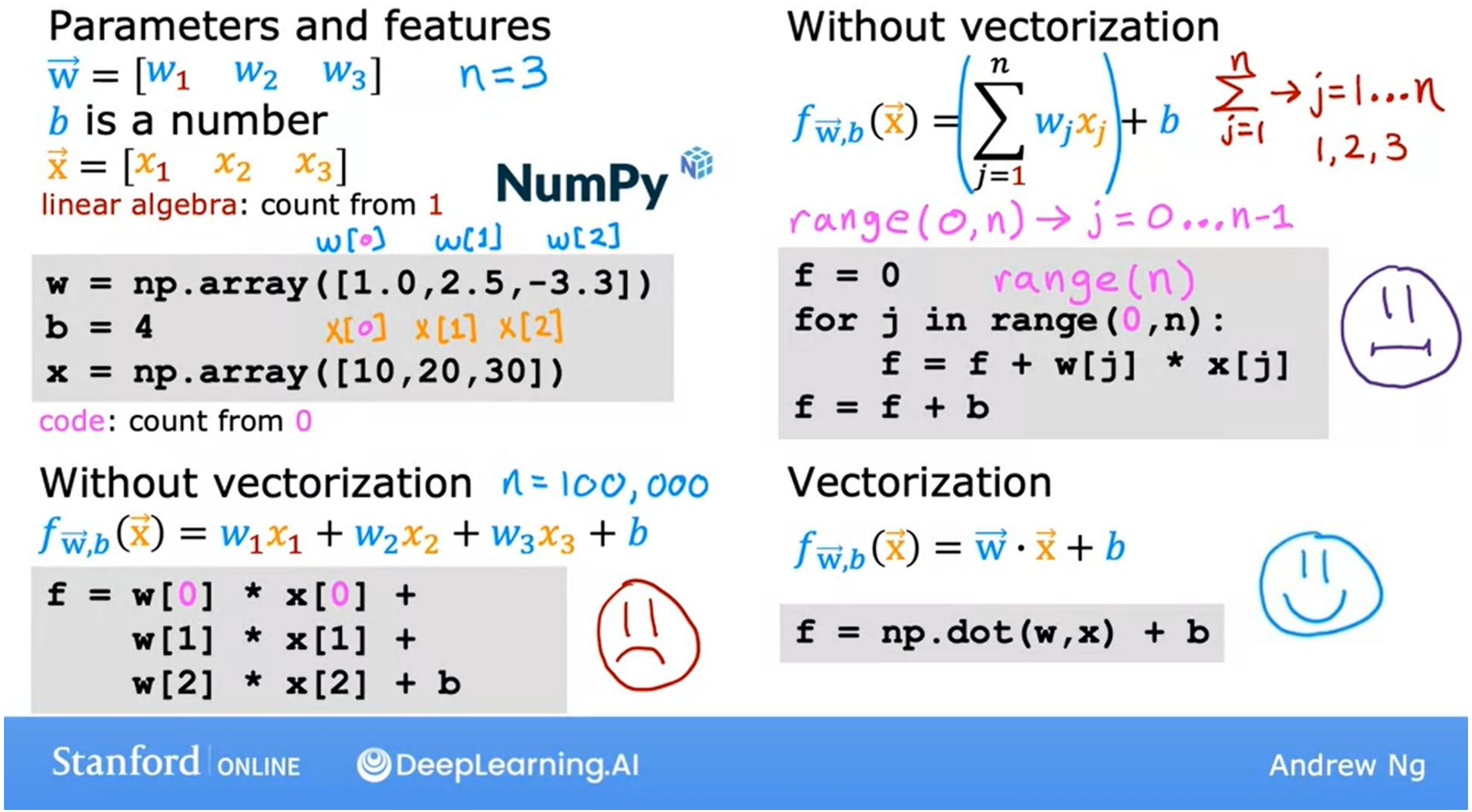
(not multivariate regression)

Vectorization



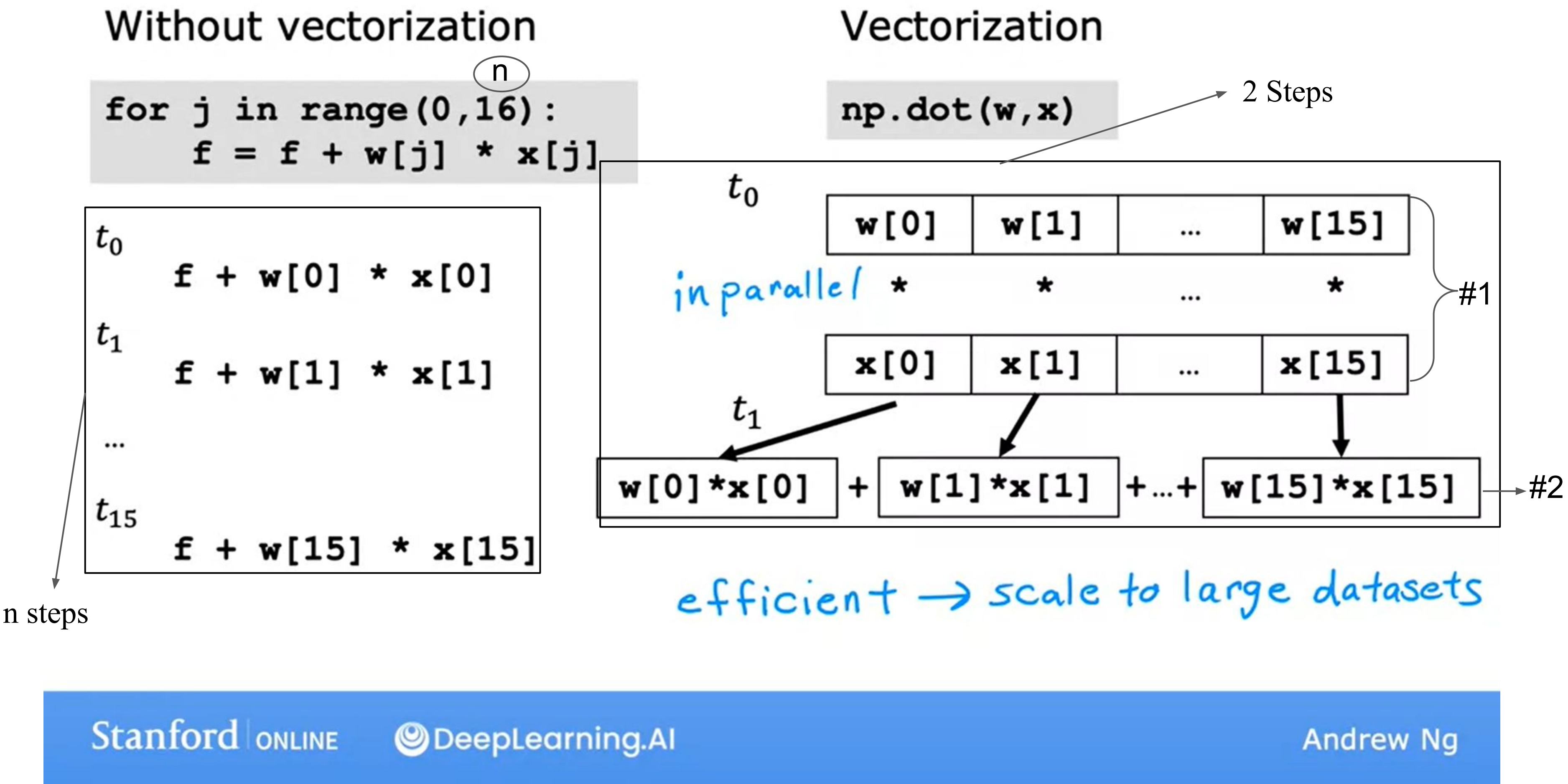
Note:
- GPU (Graphics Processing Unit) is a hardware objectively designed to speed up the computer graphics in our computer.
- GPU is often used to accelerate ml jobs.

Note:
- In linear algebra, the count starts from 1, but starts from 0 in python.



Note that numpy dot function makes the code more efficient and faster as it is able to use parallel hardware in the computer whether we use the CPU(Normal computer) or GPU(Graphics Processing Unit).

Why vectorized code is faster?



Gradient descent

$\vec{w} = (w_1 \quad w_2 \quad \cdots \quad w_{16}) \quad \cancel{b}$ parameters

derivatives $\vec{d} = (d_1 \quad d_2 \quad \cdots \quad d_{16})$

```
w = np.array([0.5, 1.3, … 3.4])
d = np.array([0.3, 0.2, … 0.4])
```

learning rate α

compute $w_j = w_j - 0.1 d_j$ for $j = 1 \dots 16$

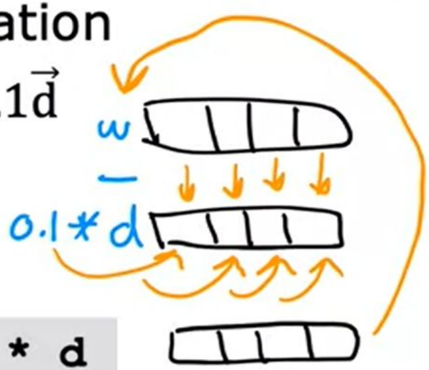
Without vectorization

$$w_1 = w_1 - 0.1 d_1$$
$$w_2 = w_2 - 0.1 d_2$$
$$\vdots$$
$$w_{16} = w_{16} - 0.1 d_{16}$$

```
for j in range(0,16):
    w[j] = w[j] - 0.1 * d[j]
```

With vectorization

$$\vec{w} = \vec{w} - 0.1\vec{d}$$




```
w = w - 0.1 * d
```

Gradient descent for multiple linear regression with vectorization

	Previous notation	Vector notation
Parameters	$w_1, \cdots, w_n$ b	vector of length n $\vec{w} = [w_1 \quad \cdots \quad w_n]$ b still a number
Model	$f_{\vec{w},b}(\vec{x}) = w_1x_1 + \cdots + w_nx_n + b$	$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$ dot product
Cost function	$J(\underbrace{w_1, \cdots, w_n}, b)$	$J(\vec{w}, b)$

Gradient descent

repeat {
$$w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(\underbrace{w_1, \cdots, w_n}, b)$$
$$b = b - \alpha \frac{\partial}{\partial b} J(\underbrace{w_1, \cdots, w_n}, b)$$
}

repeat {
$$w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$$
$$b = b - \alpha \frac{\partial}{\partial b} J(\vec{w}, b)$$
}

Gradient descent

One feature

repeat {

$$\underbrace{w}_{} = w - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{w,b}\left(x^{(i)}\right) - y^{(i)}\right) \underbrace{x^{(i)}}_{}$$

$\rightarrow \frac{\partial}{\partial w} J(w,b)$

Cost function

$$b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{w,b}\left(x^{(i)}\right) - y^{(i)}\right)$$

simultaneously update w, b

}

n features ($n \geq 2$)

repeat {

$j=1$

$$\underbrace{w_1}_{} = w_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right) - y^{(i)}\right) \underbrace{x_1^{(i)}}_{}$$

$\rightarrow \frac{\partial}{\partial w_1} J(\vec{w},b)$

$\vdots$

$j=n$

$$w_n = w_n - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right) - y^{(i)}\right) x_n^{(i)}$$

$$b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right) - y^{(i)}\right)$$

simultaneously update

w_j (for $j = 1, \cdots, n$) and b

}

An alternative to gradient descent

A complicated method used in the backend of some ml libraries.

Normal equation
- Only for linear regression
- Solve for w, b without iterations

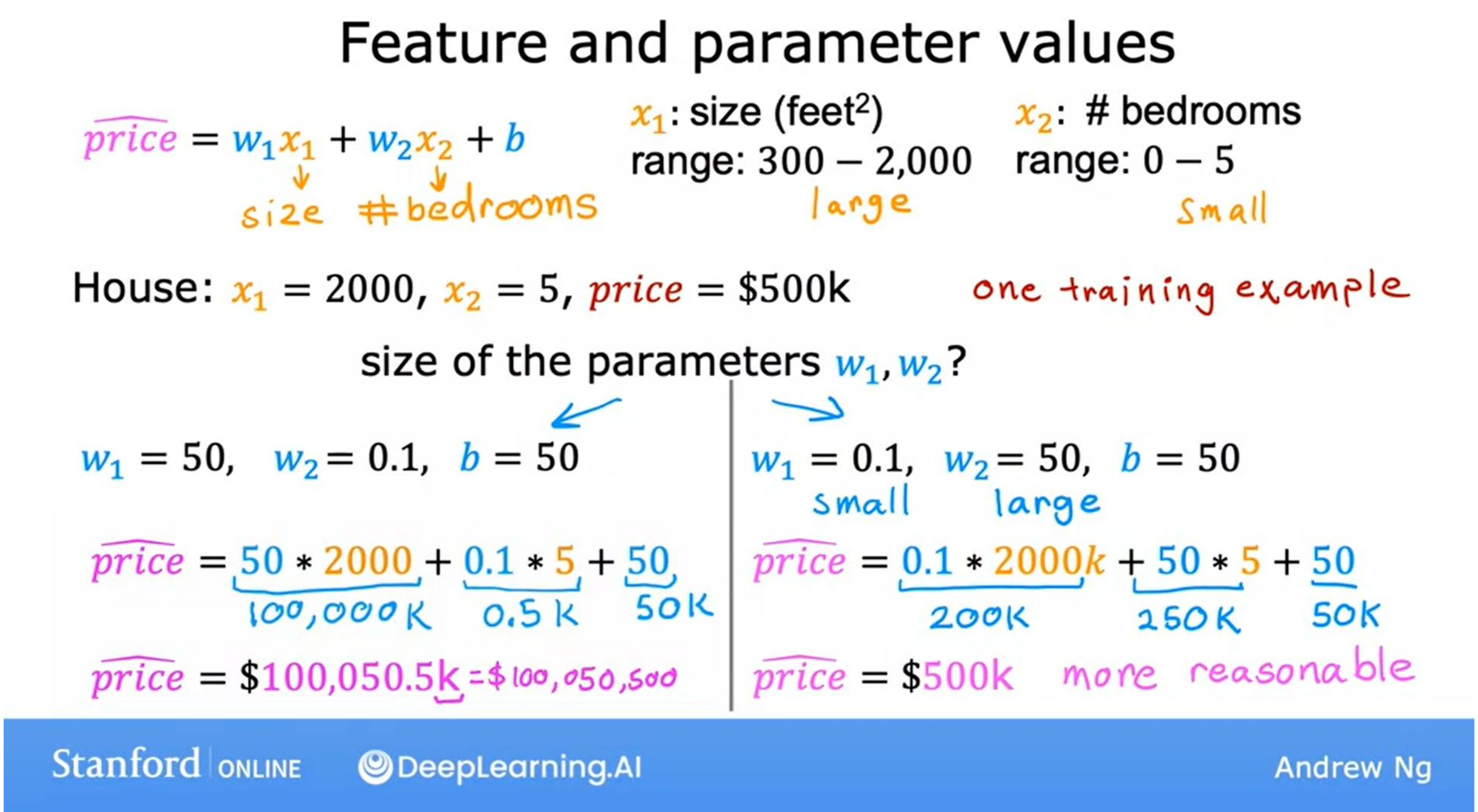
Disadvantages
- Doesn't generalize to other learning algorithms.
- Slow when number of features is large (> 10,000)

What you need to know
- Normal equation method may be used in machine learning libraries that implement linear regression.
- Gradient descent is the recommended method for finding parameters w,b

If we're using mature machine learning library and camm linear regression, there is a chance of using this method on the backend to solve w and b.

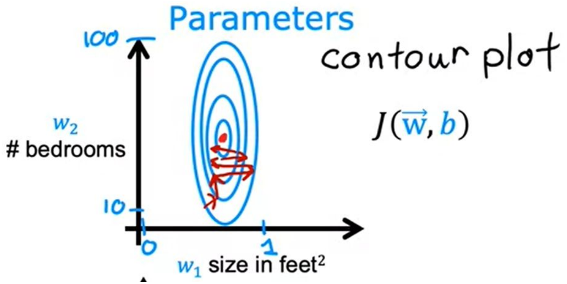
Feature scaling

- When the range of feature’s values is large, the (w) parameter will be relatively small.
- And, when the range of feature’s values is small, the (w) parameter will be relatively large.



The problem:

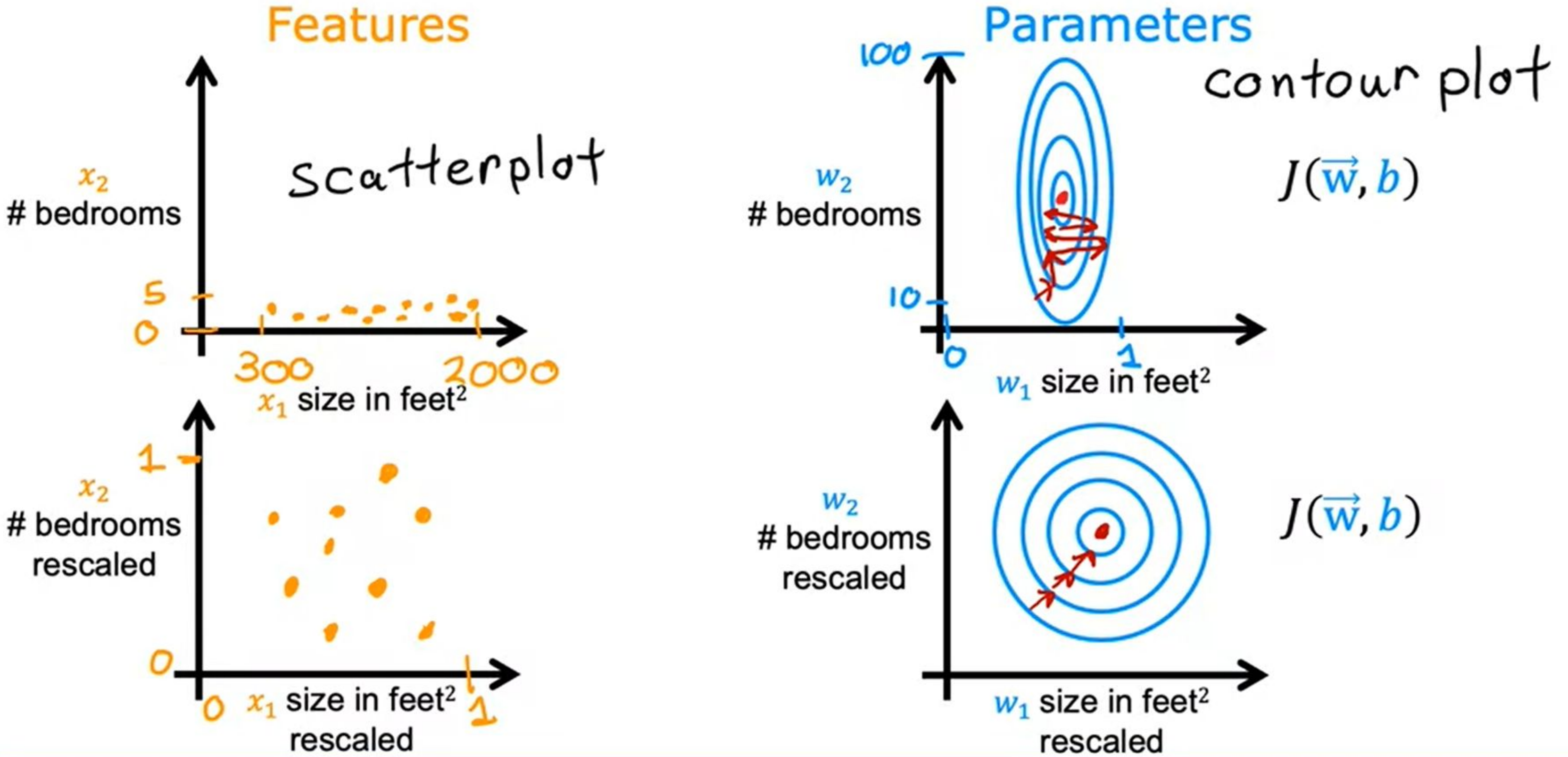
When the dataset features don’t have comparable range of values (which means that some features have large scale of values where the other features have small scale of values), the cost function will be tall and skinny like this:



Because the contours are so tall and skinny, gradient descent may end up bouncing back and forth for a long time before it can finally find its way to the global minimum. (Cause gradient descent to run slowly).

The best action here is features scaling which means performing some transformations of the training data. So, they all take on comparable range of values. Thus, we can speed up gradient descent significantly.

Feature size and gradient descent



Feature scaling methods

$$300 \leq x_1 \leq 2000 \qquad 0 \leq x_2 \leq 5$$

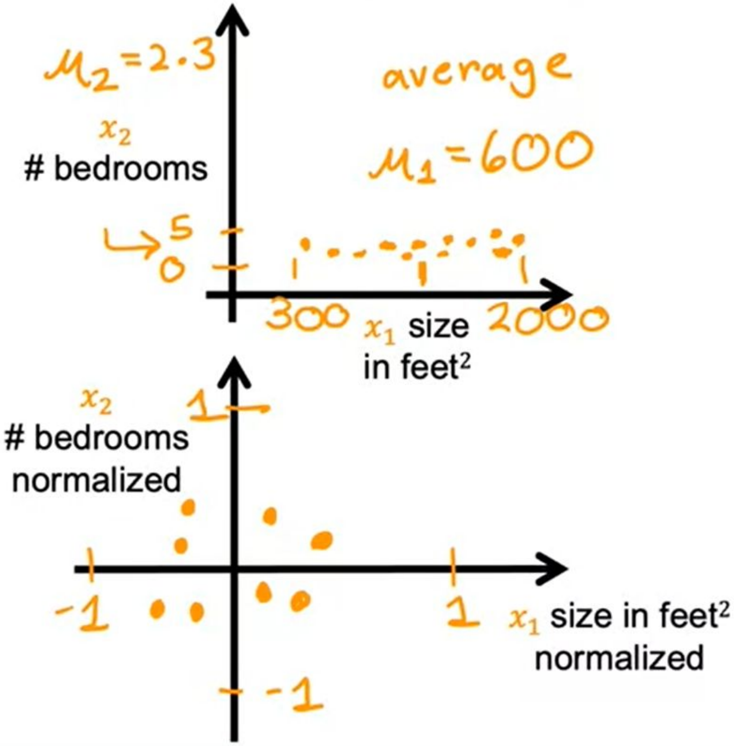
1

$$x_{1,scaled} = \frac{x_1}{\underset{max}{2000}} \qquad x_{2,scaled} = \frac{x_2}{\underset{max}{5}}$$

$$0.15 \leq x_{1,scaled} \leq 1 \qquad 0 \leq x_{2,scaled} \leq 1$$

2

Mean normalization



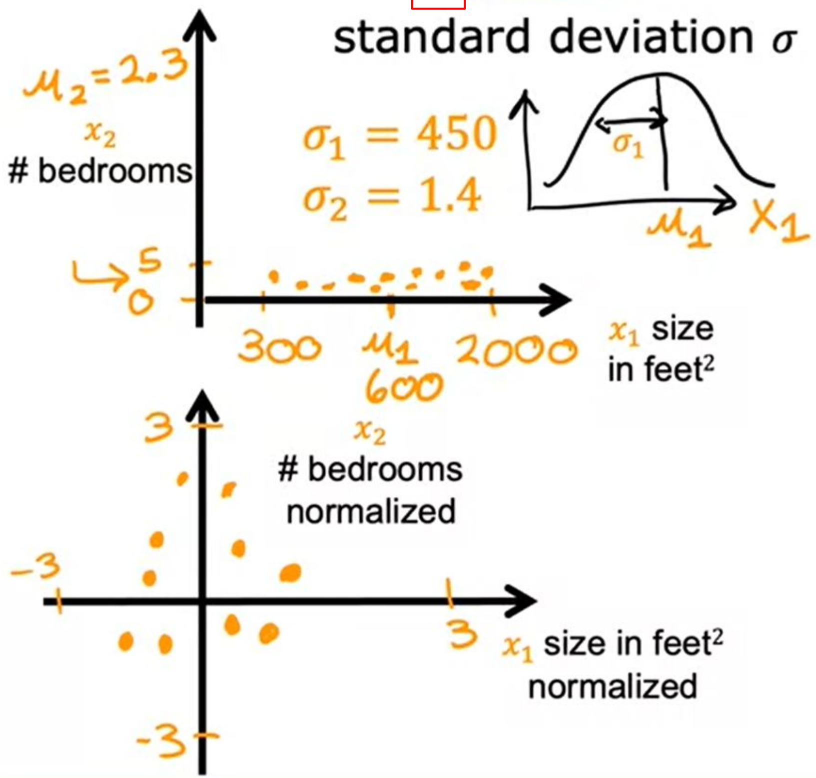
$$300 \le x_1 \le 2000 \qquad 0 \le x_2 \le 5$$

$$x_1 = \frac{x_1 - \mu_1}{\underbrace{2000-300}_{\text{max-min}}} \qquad x_2 = \frac{x_2 - \mu_2}{\underbrace{5-0}_{\text{max-min}}}$$

$$-0.18 \le x_1 \le 0.82 \qquad -0.46 \le x_2 \le 0.54$$

3

Z-score normalization



$$300 \le x_1 \le 2000 \qquad 0 \le x_2 \le 5$$

$$x_1 = \frac{x_1 - \mu_1}{\sigma_1} \qquad x_2 = \frac{x_2 - \mu_2}{\sigma_2}$$

$$-0.67 \le x_1 \le 3.1 \qquad -1.6 \le x_2 \le 1.9$$

Feature scaling

When to do feature scaling?

aim for about $-1 \leq x_j \leq 1$ for each feature x_j

$$\left.\begin{array}{c} -3 \leq x_j \leq 3 \\ -0.3 \leq x_j \leq 0.3 \end{array}\right\} \text{acceptable ranges}$$

$0 \leq x_1 \leq 3$ okay, no rescaling

$-2 \leq x_2 \leq 0.5$ okay, no rescaling

$-100 \leq x_3 \leq 100$ too large $\rightarrow$ rescale

$-0.001 \leq x_4 \leq 0.001$ too small $\rightarrow$ rescale

$98.6 \leq x_5 \leq 105$ too large $\rightarrow$ rescale

In case of doubt of scaling or not, scale it.

Make sure gradient descent is working well:
(The cost function of the training set MUST decrease over iterations as the aim is to reach the global minimum)

Learning curve

Automatic convergence test

1. Learning curve

- Plot the cost function (J), which is calculated on the training set at each iteration, iteration means after each simultaneous update of parameters (**w** & b).

- J(**w**, b) should decrease after each iteration. If it increases after each iteration, that means:

Either Alpha is chosen poorly(It usually means Alpha is too large).

Or a bug in the code.

2. Automatic convergence test

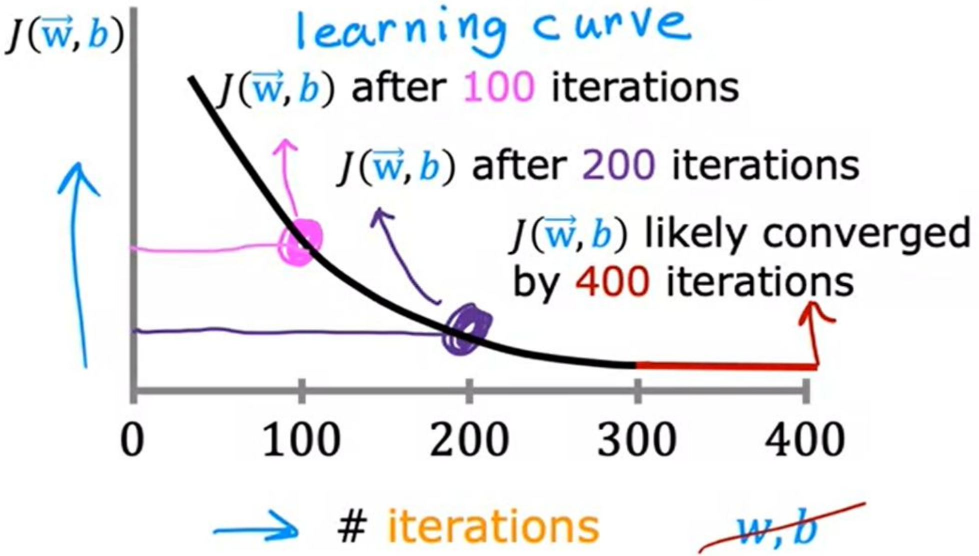
If J(**w**, b) decreases by $\leqslant \varepsilon$ in one iteration, declare convergence.

The **disadvantage** is that ε value is randomly chosen, and it is very difficult to be expected, so learning curve is better to use. In addition, learning curve provides us the ability to track the gradient descent if it is correctly working or not.

Make sure gradient descent is working correctly

objective: $\min_{\vec{w},b} J(\vec{w},b)$

$J(\vec{w},b)$ should decrease after every iteration



iterations needed varies 30 1,000 100,000

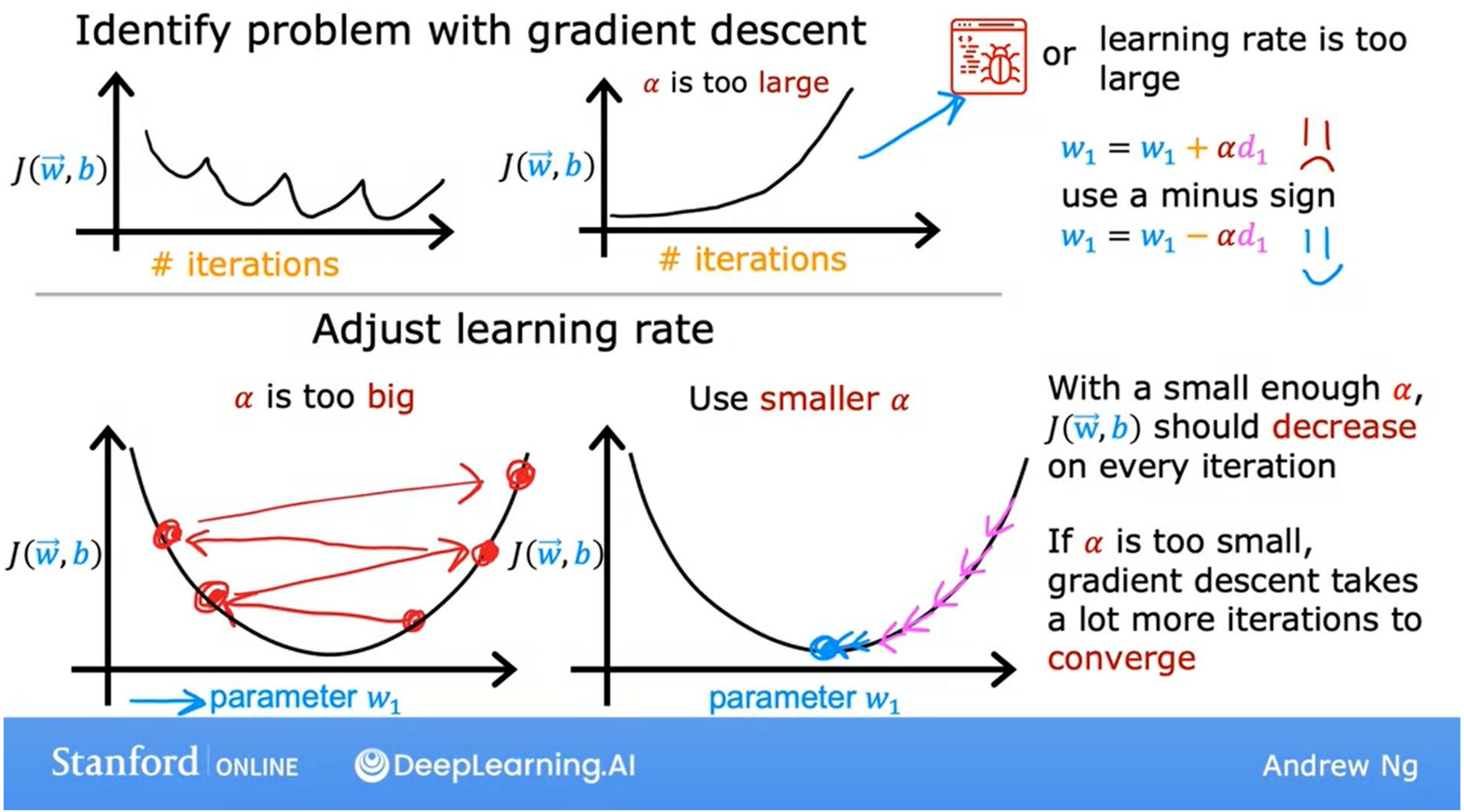
Automatic convergence test

Let ε "epsilon" be 10^{-3}.
0.001

If $J(\vec{w},b)$ decreases by $\leq \varepsilon$ in one iteration, declare convergence.

(found parameters $\vec{w}, b$ to get close to global minimum)

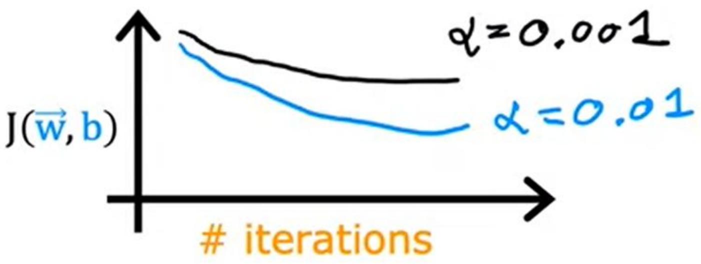
Learning rate ($\propto$)

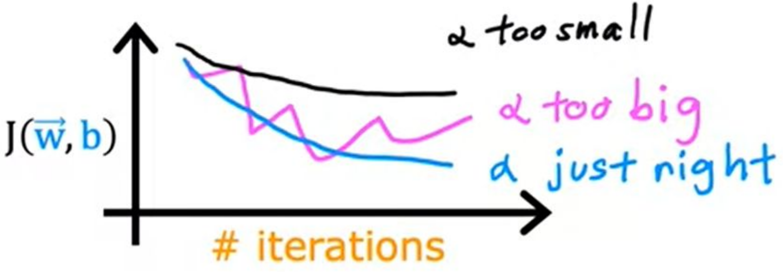


Values of α to try:

... 0.001 0.003 0.01 0.03 0.1 0.3 1 ...

3X ≈3X 3X ≈3X 3X ≈3X





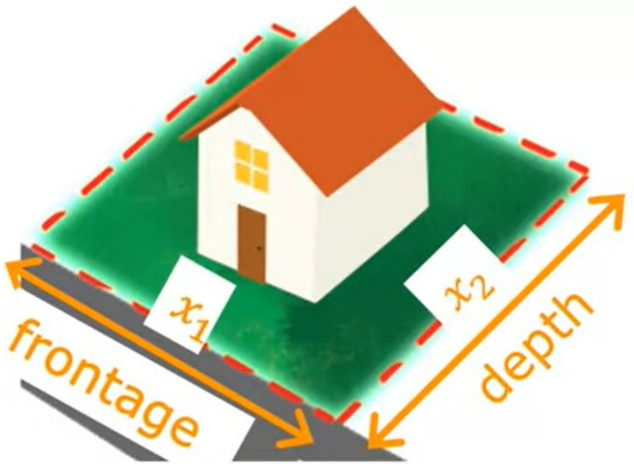
Feature engineering

$$f_{\vec{w},b}(\vec{x}) = w_1 \underbrace{x_1}_{\text{frontage}} + w_2 \underbrace{x_2}_{\text{depth}} + b$$

$$area = frontage \times depth$$

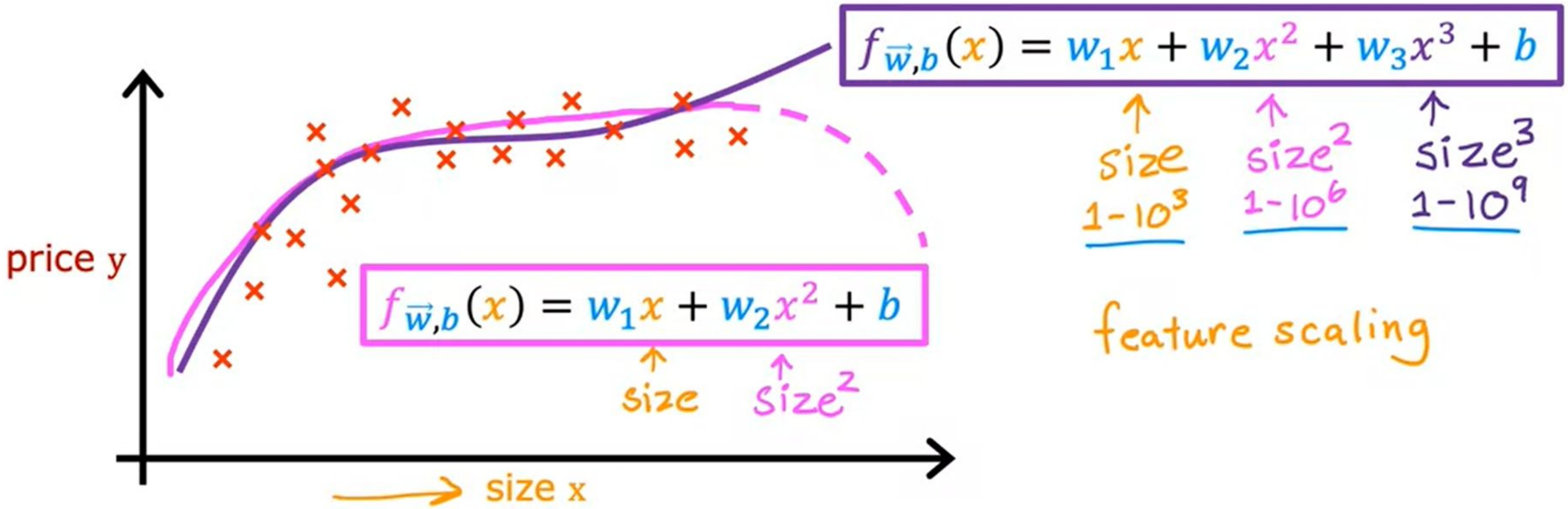
$$\underbrace{x_3 = x_1 x_2}_{\text{new feature}}$$

$$f_{\vec{w},b}(\vec{x}) = \underbrace{w_1}x_1 + \underbrace{w_2}x_2 + \underbrace{w_3}x_3 + b$$

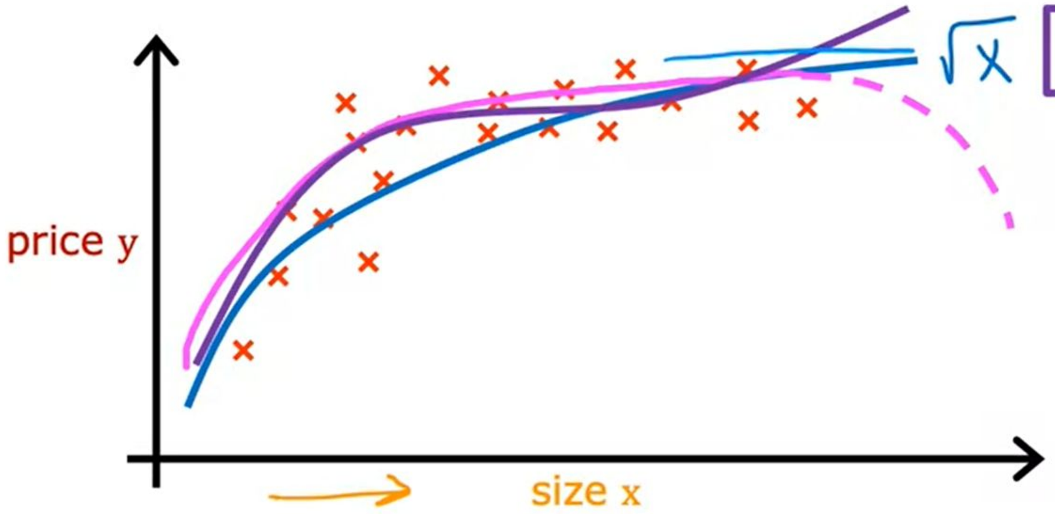


Feature engineering:
Using intuition to design new features, by transforming or combining original features.

Polynomial regression



Choice of features



$$f_{\vec{w},b}(x) = w_1 x + w_2 \sqrt{x} + b$$

size

$\sqrt{\text{size}}$

what features to use?

↪ course 2