Supervised Learning Notes: (1)

Supervised learning is the most commonly used machine learning technique. About 99% of today's ML algorithms use supervised learning.

In supervised learning, we provide the computer with training data that includes both inputs (x - features) and outputs (y - labels). Given specific feature inputs, the algorithm learns to generate the corresponding output value.

Types of supervised learning:

1. Regression
2. Classification

Regression:

In regression, we create a mathematical relationship between features and labels. This relationship helps us correlate features with their corresponding labels.

With regression, given a feature X, we predict its label Y.

Example: Given a house size, predict its value.

Classification:

In classification, the computer creates clusters of data, which we can label to form categories. This allows us to classify data into different groups.

Examples: Image scanning to detect cancer, self-driving cars distinguishing between humans and vehicles.

Linear regression equation:

In linear regression, we represent the relationship between features and targets using an equation. Features are values provided by training examples, while targets are values the function needs to predict.

Feature is represented by x and target is represented by y.

The number of training examples for a feature and target value is represented by **m**.

The ith training example is represented by **x_i, y_i**

The general equation for linear regression with a straight line is:

f(x) = wx+b, where w is the weight of the feature and b is the bias (distance from actual value)

f(x) represents the target value y. Y^ denotes the estimated value of y.

Cost functions:

Cost functions measure the difference between predicted and actual values.

W and b represent coefficients or weights that we calculate during training to ensure our predictions are as close as possible to actual values.

The difference between actual and predicted values is called error. Cost functions are typically an average of summed squared errors. The formula is:

Mean Squared Error (MSE) cost function:
J(w,b) = 1/2N (sum of (predicted value - actual value)^2)

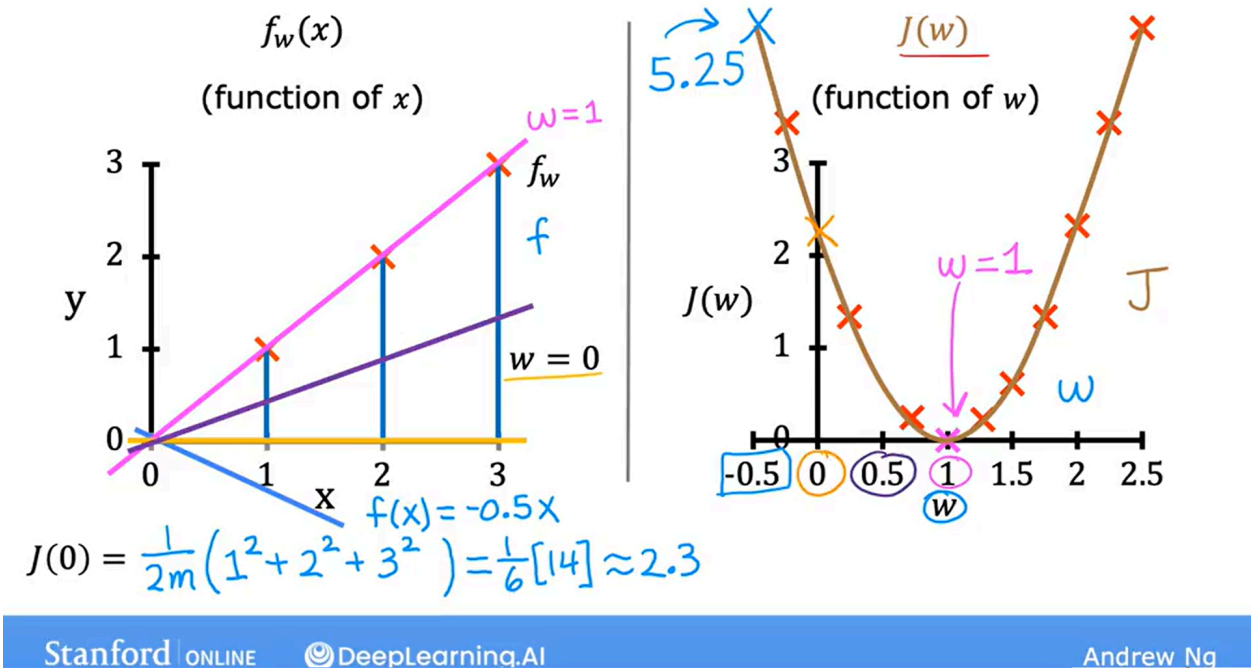
N = total number of training examples/data points

We use 2N in the denominator to simplify the derivative calculation in the cost function.

The cost functions are calculated based on coefficients w, b and the goal is to choose w, b such that we get minimum MSE from cost function.

For simplicity, we often consider b = 0 (meaning y = 0 when x = 0). When we plot f(w,b) = wx + b, it becomes just a function of weight w: f(w) = wx. The value of f(w) (predicted value) depends on feature 'x' for a constant value of w.
Similarly, the cost function J(w,b) simplifies to J(w), where the cost function depends only on weight w.

In the following graph, we can see that f(w) creates different line graphs for various values of w, with corresponding points in the cost function for each value of w.



The goal is to select a value of w such that cost function J(w) results in minimum cost.

Summary:

Model = f(w,b) = wx + b

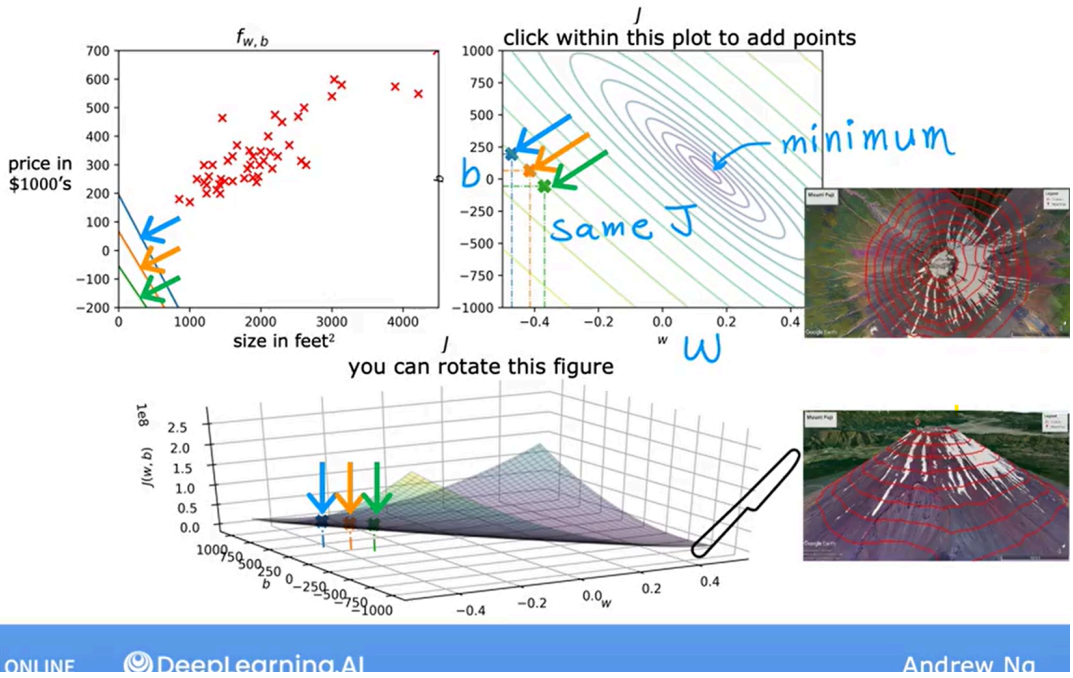
Parameters = w,b

Cost function = J(w,b) = 1/2N(sum of (predicted value - actual value)^2)

Objective = Minimize J(w,b)

Model	$f_{w,b}(x) = wx + b$
Parameters	w, b
Cost Function	$J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}(f_{w,b}(x^{(i)}) - y^{(i)})^2$
Objective	$\underset{w,b}{\text{minimize}}\, J(w,b)$

Now in real world scenario, the value of parameter b won't be 0. Hence, the cost function will now look like a 3-d shape instead of 2-d because it will be dependent on 2 parameters.



As we can see 3-d printed graph of cost function J(w,b) is in form of a bowl. Now the minimum value of J(w,b) will be at the lowest point of the bowl. In order to

imagine the bowl as 2-d graph, we draw a contour map. Just like we visualize a mountain from top and draw horizontal lines to represent different heights on the mountain, we draw contour of 3-d graph.

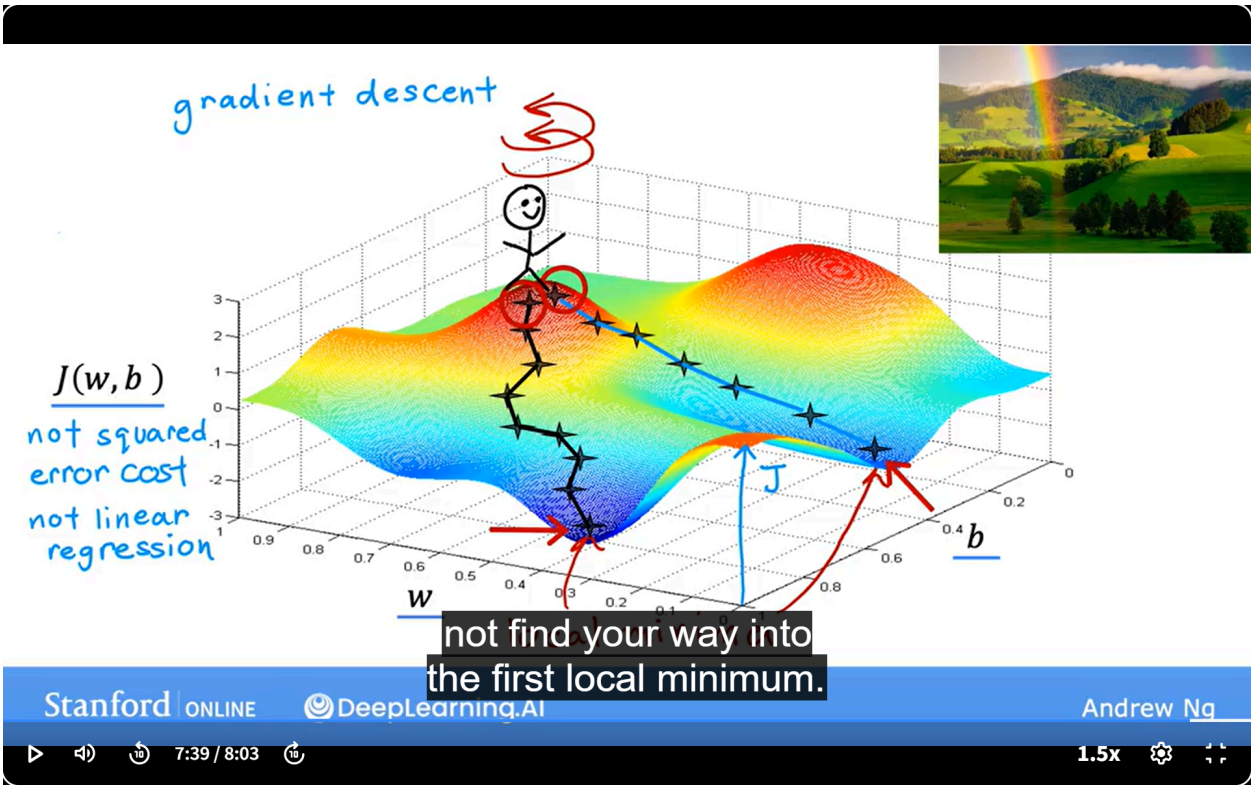
In contour graph, the minimum value of J(w,b) lies at the smallest concentric circle. A circle in contour represent a constant J(w,b) value for different w and b parameters.

Gradient Descent:

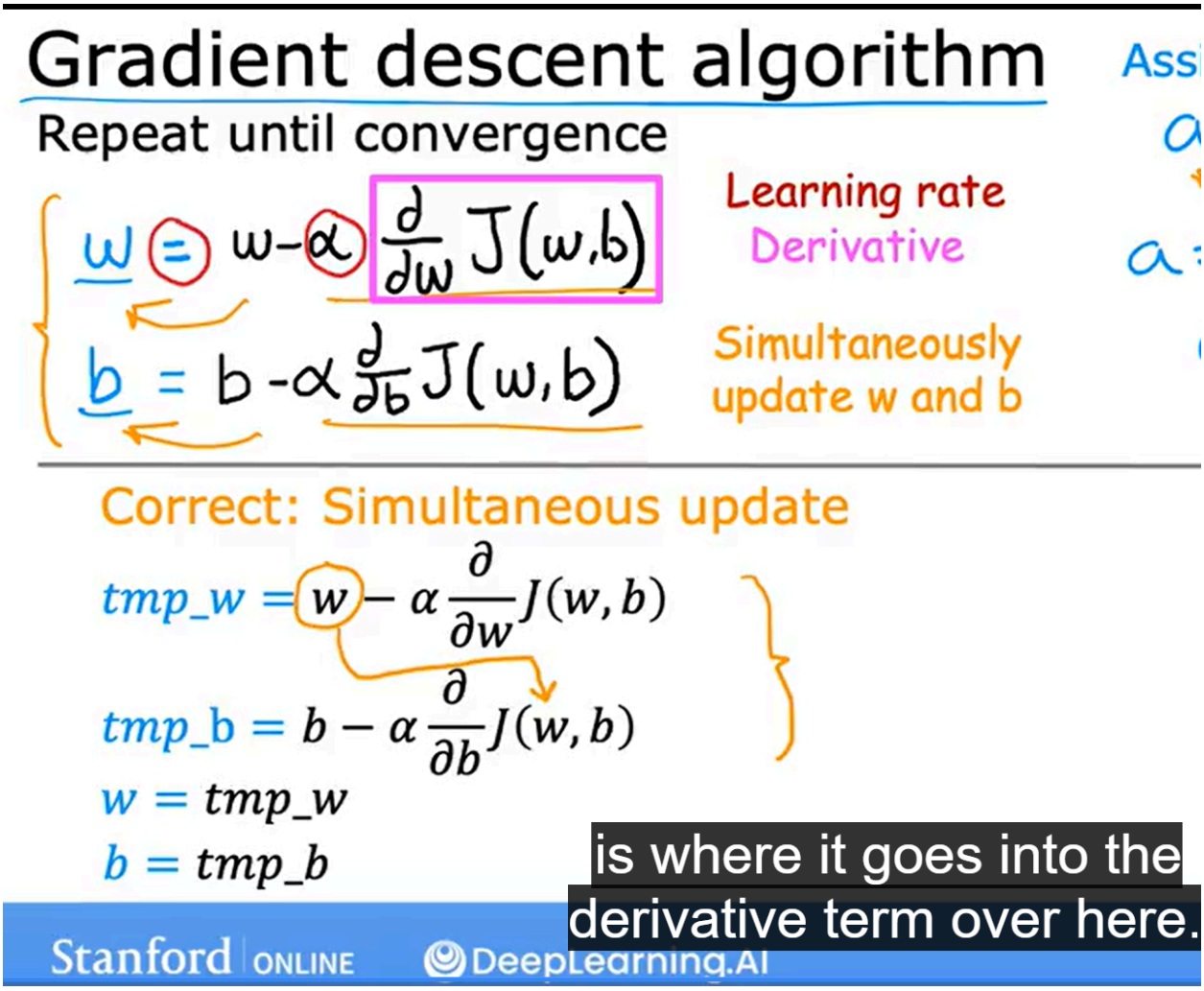
Gradient descent is an approach that helps us find values of parameters w and b that result in minimum cost. While the graph we saw earlier for linear regression is simple, real-world scenarios often have multiple maxima and minima (similar to a golf course).

For complex graphs with multiple parameters, predicting the optimal w and b values becomes challenging. This is why we use gradient descent to find minima.

Gradient descent works by taking multiple paths downward from a single point in different directions to identify local minima.



General gradient descent algo looks like this, where we calculate w, b parameters using learning rate - alpha, and multiply it with derivative of cost function J(w,b)

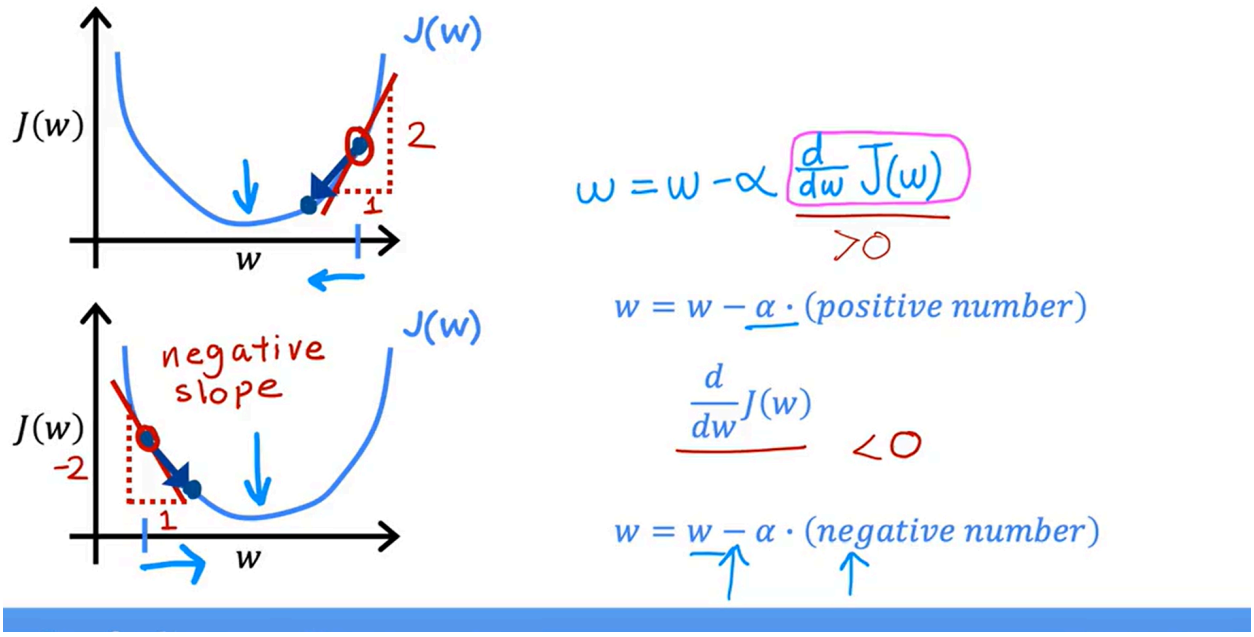


The following graphs illustrate how gradient descent works. A derivative represents a point on the graph, which can be calculated by drawing a tangent and measuring its slope (J(w)/w).

When the slope is positive, the derivative is positive, causing the newly computed value of w to be less than its previous value. This means w will continue decreasing and move toward the cost function's minimum.

Similarly, when the slope is negative, the derivative is negative, making the newly computed value of w greater than its previous value. The value of w will continue increasing until it reaches the cost function's minimum.

In this way, gradient descent ensures that with each update of parameter w, we consistently move toward the cost function's minimum.

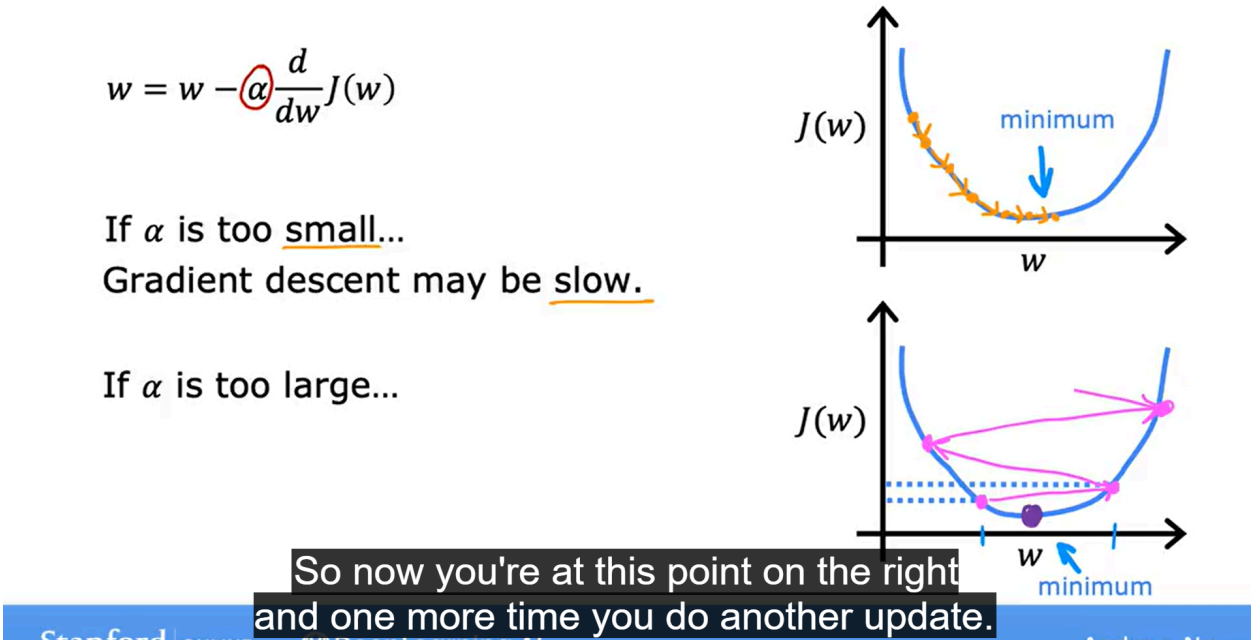


The learning rate (alpha) represents the step size we take toward the next value of w and b. A small alpha results in minimal parameter changes, meaning we'll reach the minima over a longer period. However, with a small learning rate, we can be confident that we'll reach the minima and converge at the minimum cost function.

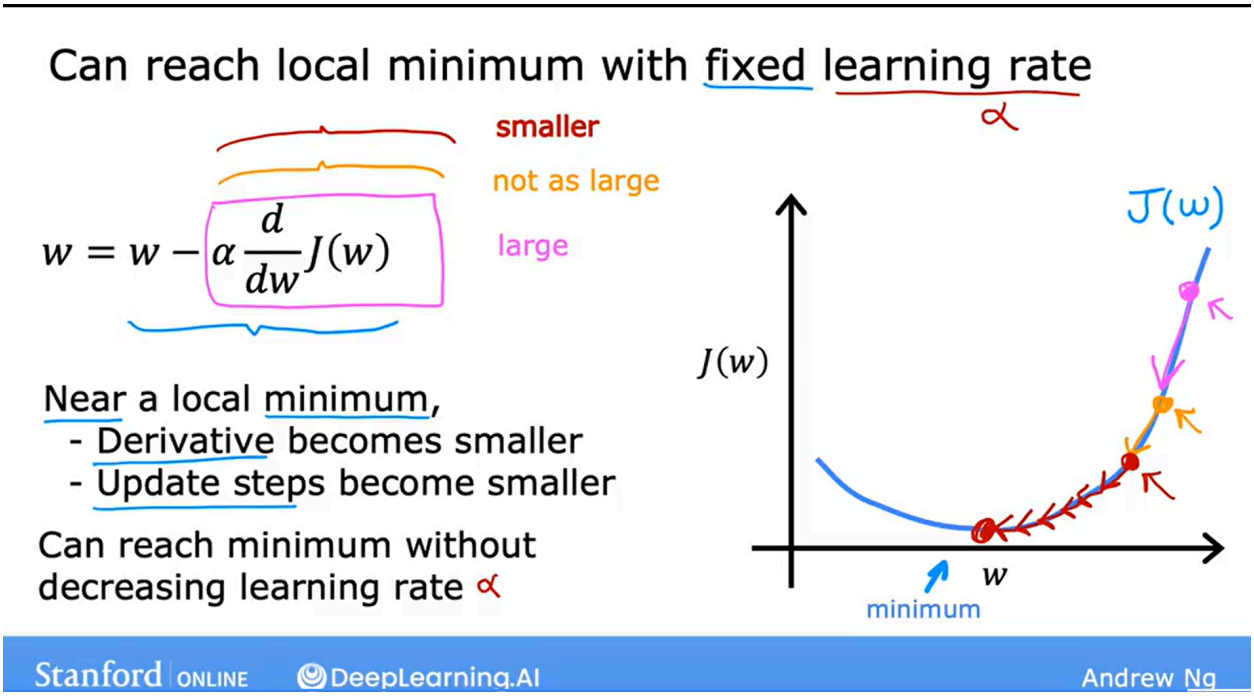
In contrast, a large learning rate causes significant parameter changes, potentially resulting in major variations that might miss the minima. Instead of converging, we might start diverging from the minima.

For example, in the diagram below, we can see a cost function J with only parameter w. With a small learning rate, the change in w's value is small, so it reaches minima slowly.

Whereas with a large learning rate, we see significant changes in newly computed values of w, which can cause the algorithm to miss the minima and diverge.



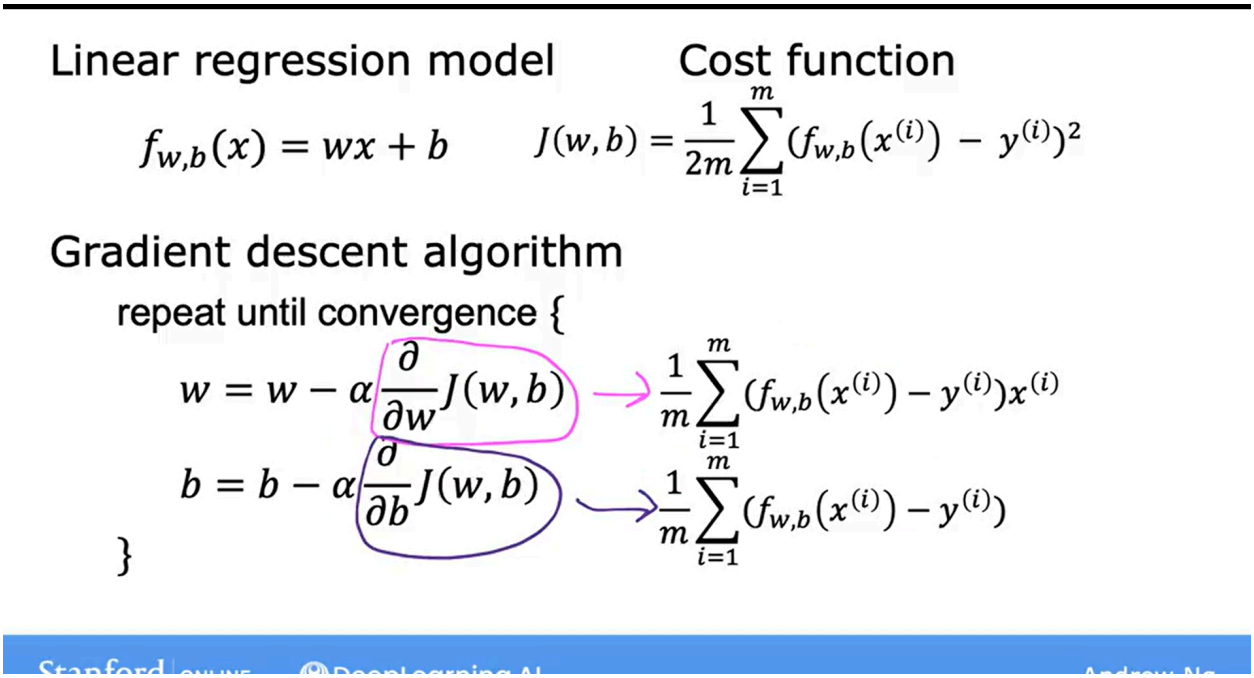
As we reach local minima we don’t need to change the value of the learning rate. With a constant learning rate, we can see that as we reach towards minima, the value of derivative decreases which means the updates becomes smaller and smaller.



Conclusion:

In conclusion, we learned 3 basic functions:

1. Linear regression: To calculate a label/target y based on a feature x.
2. Cost function: To calculate model accuracy by comparing predicted value f(x) with actual target y.
3. Gradient descent: To calculate values of parameters w and b so that we can get minimum cost function value..



Regression with multiple linear regression:

Previously, we examined linear functions with a single feature, where we assign a weight and calculate the label y.

In real-world scenarios, predicting house prices involves multiple features such as location, year built, and number of rooms. We need to incorporate these values and assign them appropriate weights.

Linear regression with multiple variables can be represented as:

y = w1×1 + w2×2 + w3×3 + ... + wnxn + b

Each feature is assigned a unique weight to predict the house price y. These multiple features and weights can be represented as vectors (arrays), allowing us to calculate the equation using the dot product of these arrays.

Without vectorization

$$f_{\vec{w},b}(\vec{x}) = \left(\sum_{j=1}^{n} w_j x_j\right) + b$$

$\sum_{j=1}^{n} \rightarrow j = 1 \ldots n$ (1, 2, 3)

range(0, n) → j = 0 ... n-1

```
f = 0          range(n)
for j in range(0,n):
    f = f + w[j] * x[j]
f = f + b
```

Vectorization

$$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$$

```
f = np.dot(w,x) + b
```

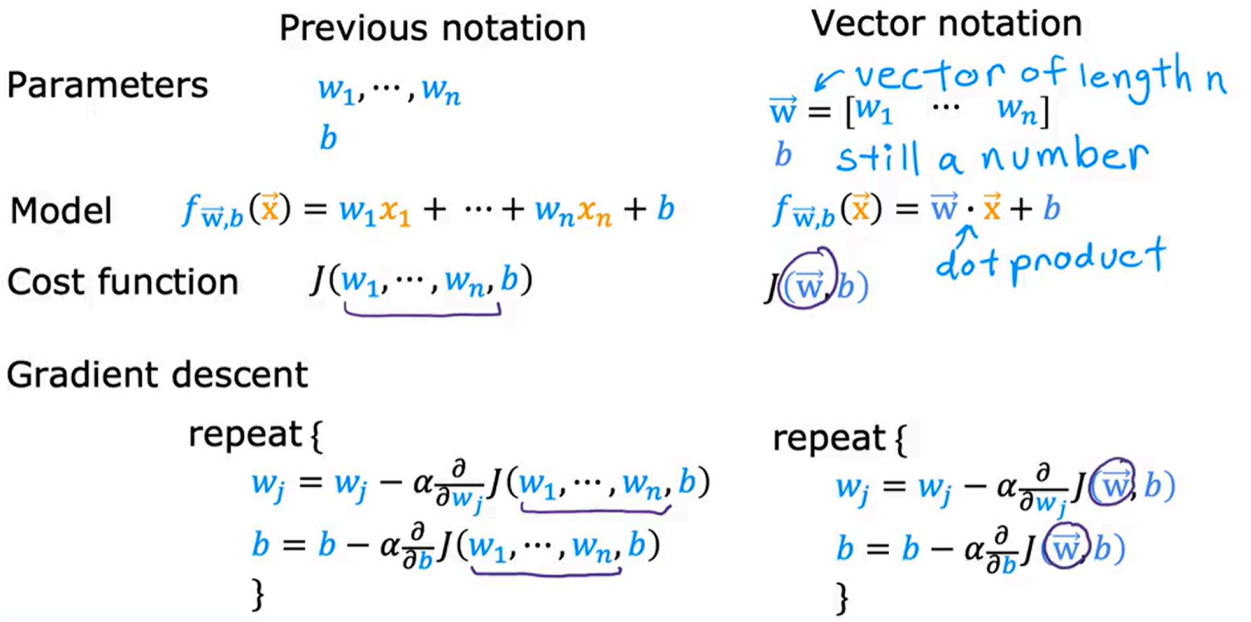
Without vectorization, products are calculated sequentially, which works fine with a small set of features. However, with a large number of features (such as billions of parameters), we need a more optimized approach. This is where vectorization dot product comes in, calculating these products in parallel on a GPU.

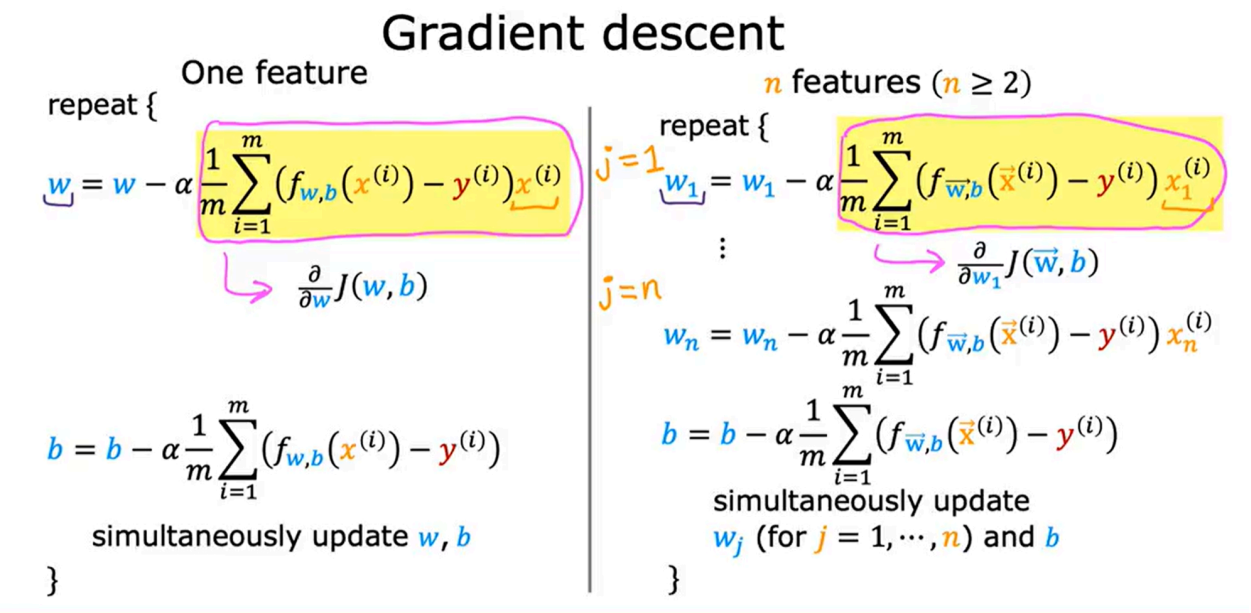
With multiple features, our equations will look something like this: w is now a vector with n values in the array, b remains the same, and x is another vector. The model function F(x) will now have the dot product of w and x.

For gradient descent with multiple features, we calculate the **derivative of cost function (the gradient)** by taking the summation of dot products of w and x over m iterations, where **m = number of training examples.**

We repeat this process for each of the **n weights** (n = number of features) and update each **Wi** with each new iteration.

(For linear regression, we use all examples in our training set rather than a partial training set).





Alternative to gradient descent:

Alternative to gradient descent:

A **Normal Equation** provides an alternative to gradient descent. This method calculates w and b directly without iterations, making it faster in certain cases. However, it has two main disadvantages: it only works for linear regression (not for other models like logistic regression), and it becomes computationally expensive when dealing with large feature sets (n > 10k).; 10k.

Look at file Linear_Regression_MultipleVariable jupyter notebook for implementation

Feature Scaling:

Feature scaling is used for scenarios where multiple features in linear regression have range of values which are far apart. For eg.

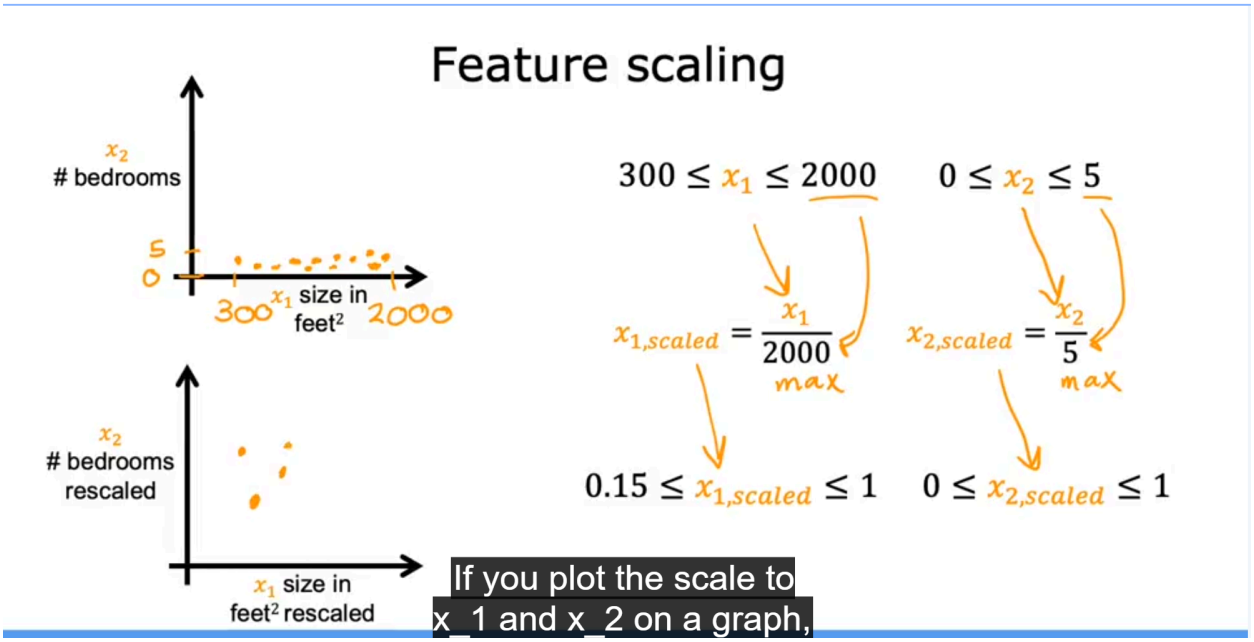
$y = w1 \times 1 + w2 \times 2 + b$

where x1 = square feet size of house and x2 = number of bedrooms.

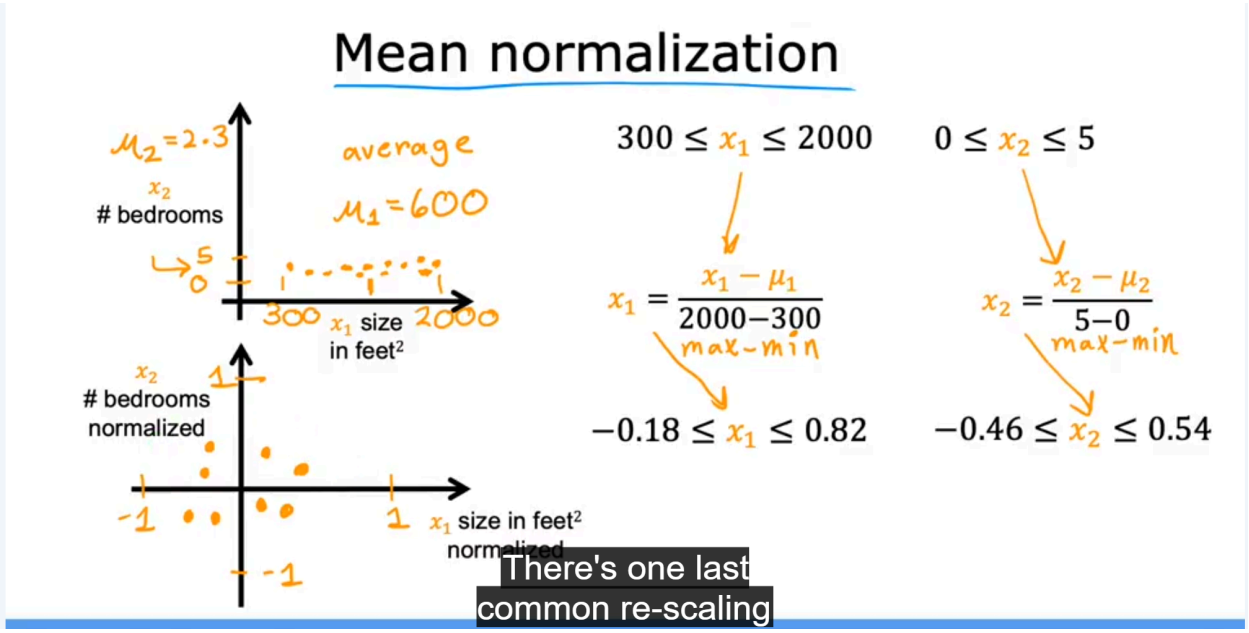
Generally, range of x1 will be between 300 and 10k and x2 can be between 0-5. This difference in the range of feature values is huge due to which it will be difficult to plot graph of cost between these features. Hence gradient descent will take a longer time to find the global minima.

Hence, in order to optimize time for gradient descent we try to shift two feature values in near by range so that they are comparable and gradient descent can quickly find minima.

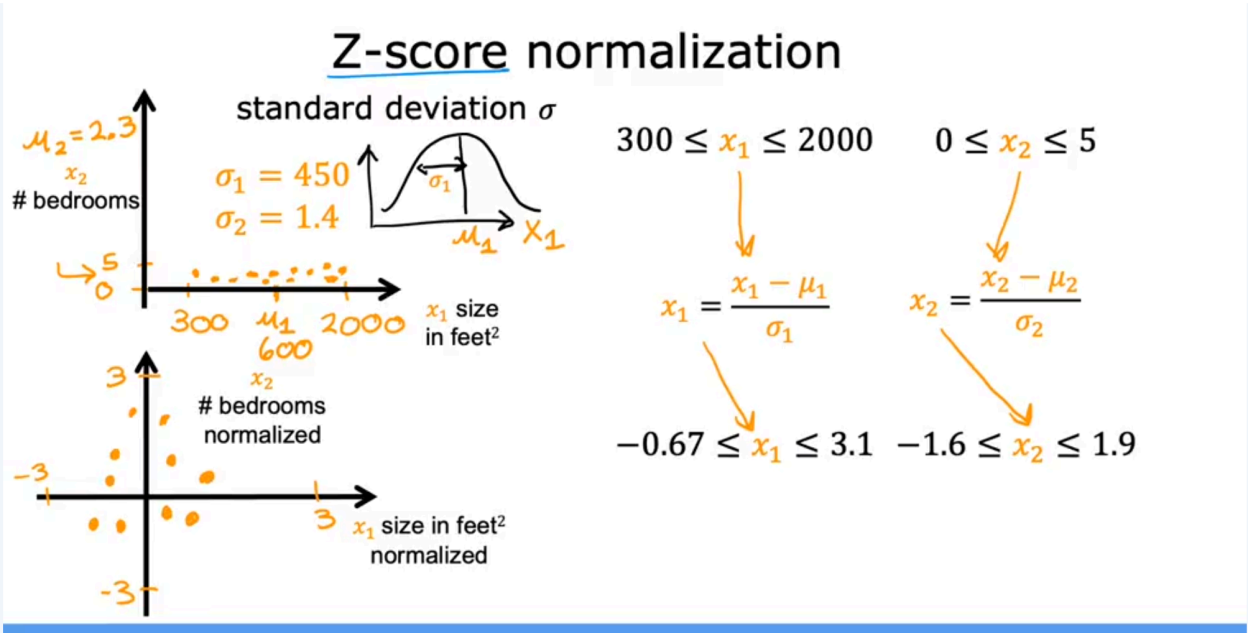
There are different types of feature scaling. In a simple feature scaling we divide the feature value by its maximum which will give us range between 0-1. Hence both the feature values are now in comparable range



Similarly there is **mean normalization**, where we scale feature by first calculating the average then take the different between feature value Xi and average and divide that by difference between max and min feature value.



Another type of scaling is **Z-score scaling,** which takes standard deviation into account. In Z-score scaling instead of dividing by difference between max and min feature value, we divide it by standard deviation.



Checkout FeatureScaling_ZScore jupyter notebook for more info and implementation.

Polynomial Regression:

Polynomial Regression is used for plotting models where a straight line can't fit. Hence instead of linear we use quadratic or polynomial equation. For eg.

f_wb = w1.x + w2.X^2 + w3.X^3

The graph for this equation won't be a straight line instead it will curve and might be suitable for multiple feature scenario.

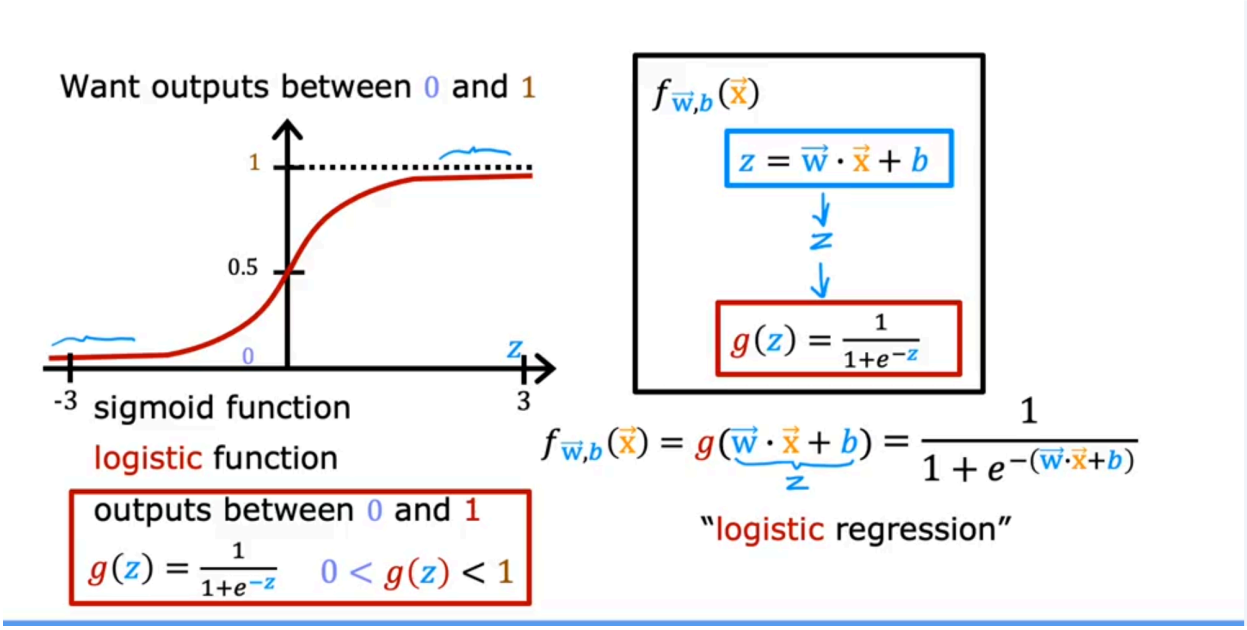
Although, the difference between the range of each feature will be huge since x < x^2 < x^3 which in turn will increase time for computing gradient descent. Hence for such polynomial regression equations, we first need to scale the features by using z-score normalization so that they are in comparable range and then run gradient descent to calculate minimum cost and w, b parameters.

Checkout FeatureScaling_PolynomialRegression Jupyter notebook for implementation.

Classification:

In classification we can not use linear regression since linear regression predicts numerical values but here we are trying to classify objects into categories. Hence we use Logistic Regression where we use sigmoid function to divide data into category 0, 1

Formula for logistic regression is below:



Here 'e' is a mathematical constant = 2.7. The value of g(z) i.e. sigmoid function always lies between 0 and 1.

Logical regression divides the input parameters into 2 categories - 0 or 1, based on the value of f_wb.

As we see f_wb = g(z) where z is the model function = wx + b (for linear regression scenario).

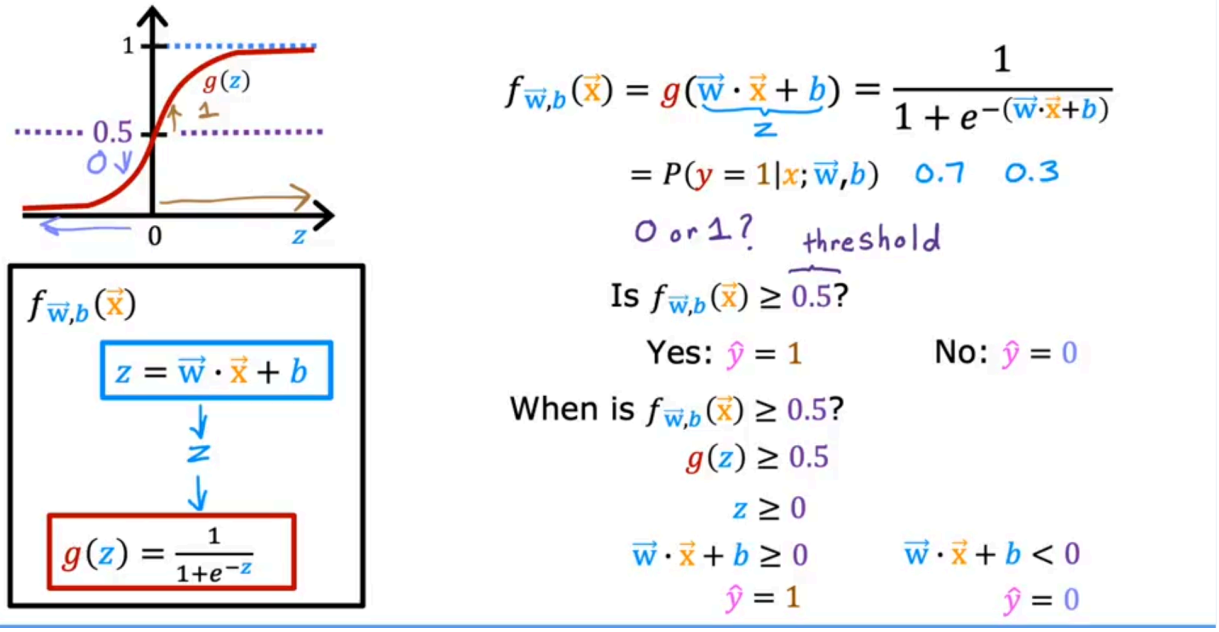
Hence we create a decision boundary where value < boundary ⇒ 0 and value > boundary ⇒ 1

As we see f_wb depends on g(z) which in turn depends upon z. Hence to create a boundary we check if z ≥ = 0 that means predicted_value ⇒ 1 else predicted_value ⇒ 0.

Z can be any model function (not just linear regression) and we can adjust w, b such that it will impact decision boundary value. For eg.

Z = w1×1 + w2×2 + b

lets say w1 = 1, w2 = 1, and b = -3 that means the Z > 0 when x1+x2 > 3 which means our decision boundary lies at y axis at 3.



Loss function in Logistic Regression:

A loss function in logistic regression is similar to cost function in linear regression.

In linear regression we used to calculate cost function by taking mean of squared errors i.e. difference between predicted and actual values.

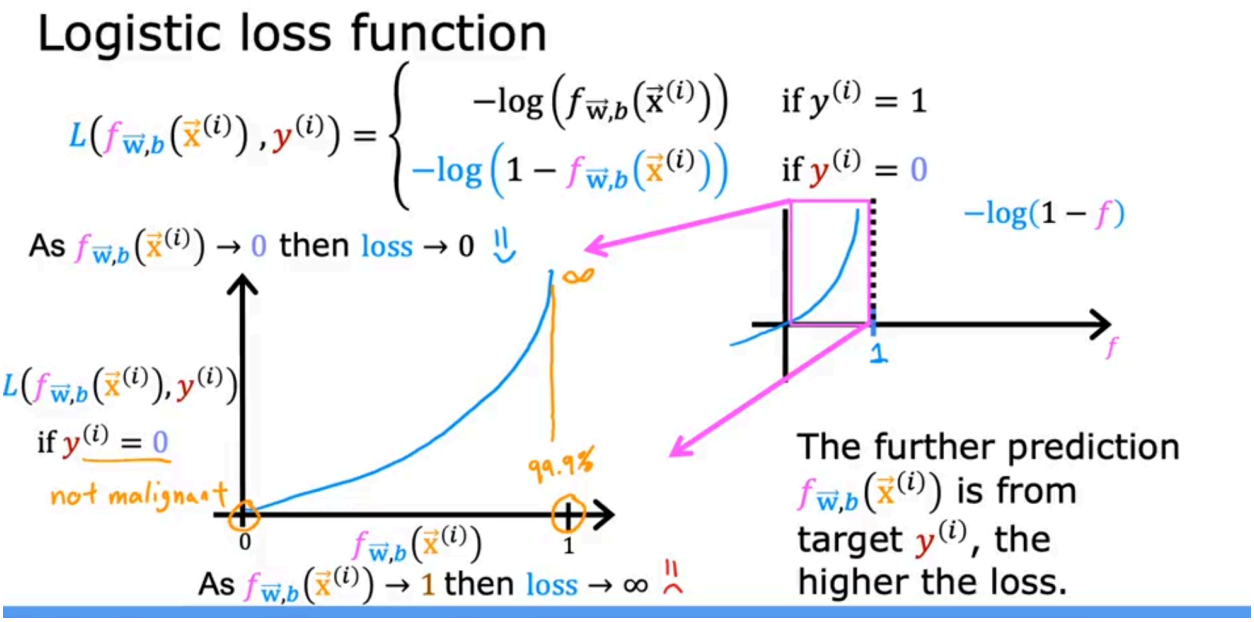
If using the mean squared error for logistic regression, the cost function is "non-convex", so it's more difficult for gradient descent to find an optimal value for the parameters w and b.

In loss function, since the actual values of y is only 0 or 1 hence we can't take the difference between predicted and actual as a cost. Instead we use log function.

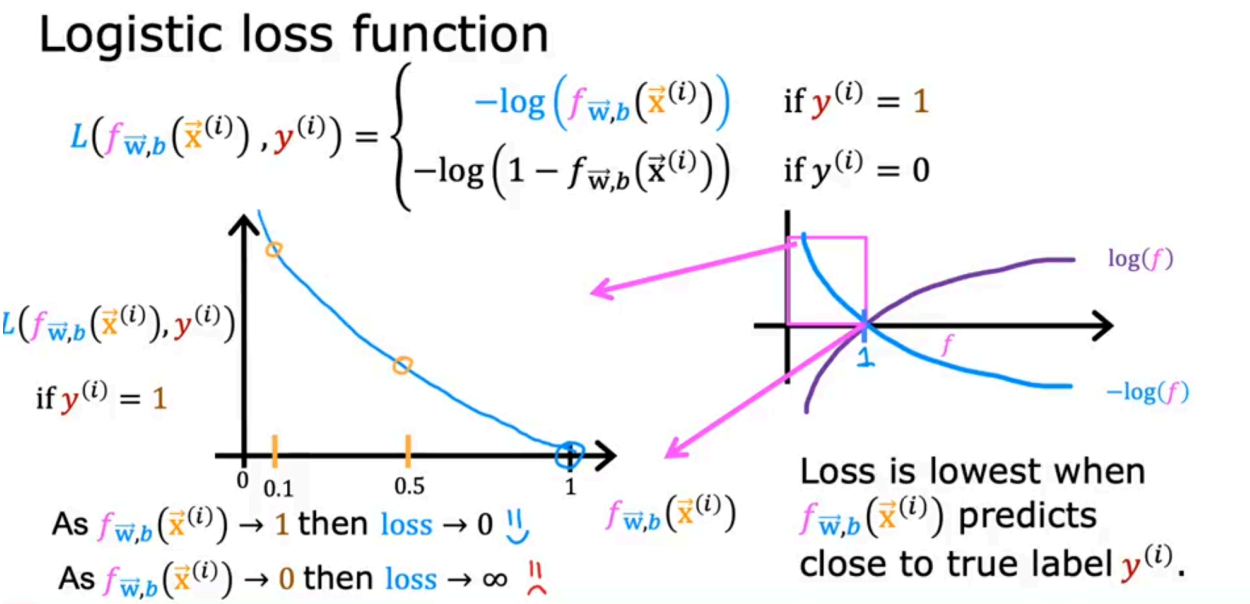
If y = 1 then loss function = -log(f_wb)

if y = 0 then loss function = -log(1 - f_wb)

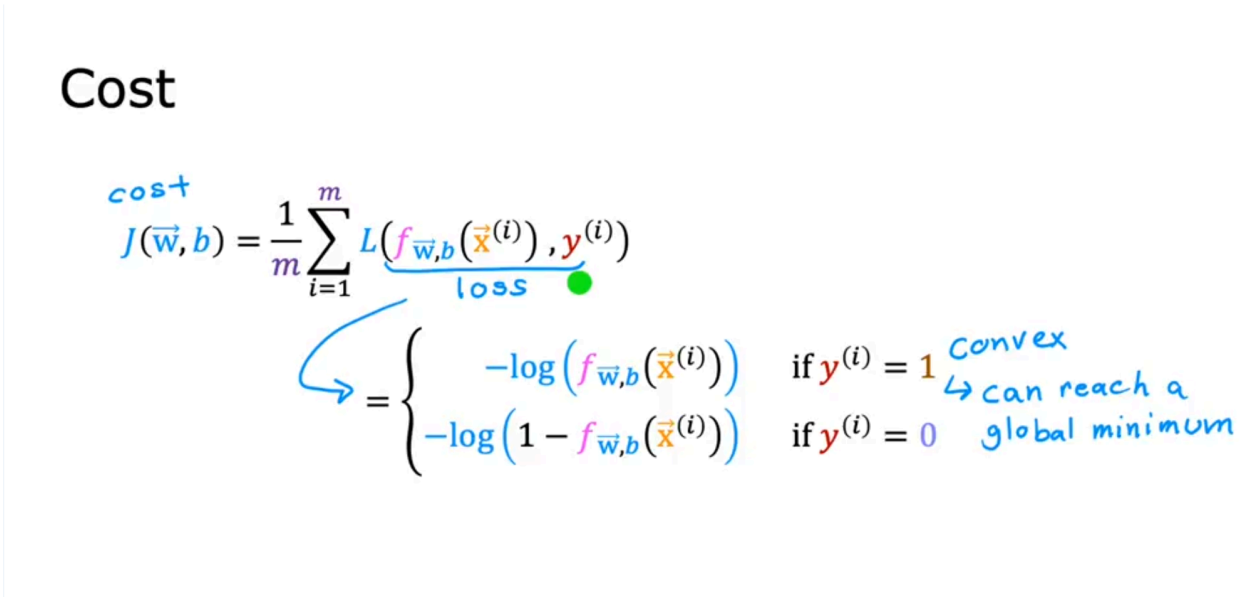
For y = 0 the graph of loss function can be represented as (x-axis - predicted value, y-axis - loss function):



Similarly for y =1 the graph of loss function can be represented as (x-axis - predicted value (f_wb), y-axis - loss function (-log(f_wb)).



Hence cost in logistic regression can be represented as sum of loss function over m training examples:



To simplify loss function, we can write it as a single equation as:

Simplified loss function

$$L\left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right), y^{(i)}\right) = \begin{cases} -\log\left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right)\right) & \text{if } y^{(i)} = 1 \\ -\log\left(1 - f_{\vec{w},b}\left(\vec{x}^{(i)}\right)\right) & \text{if } y^{(i)} = 0 \end{cases}$$

$$L\left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right), y^{(i)}\right) = -y^{(i)}\log\left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right)\right) - (1 - y^{(i)})\log\left(1 - f_{\vec{w},b}\left(\vec{x}^{(i)}\right)\right)$$

0 $\qquad$ $(1-0)$

if $y^{(i)} = 1$:

$$L\left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right), y^{(i)}\right) = -1\ \log\left(f(\vec{x})\right)$$

if $y^{(i)} = 0$:

$$L\left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right), y^{(i)}\right) = -\ (1-0)\log(1 - f(\vec{x}))$$

And similarly, the cost function can be simplified as:

Simplified cost function

loss

$$L\left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right), y^{(i)}\right) = -y^{(i)}\log\left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right)\right) - (1 - y^{(i)})\log\left(1 - f_{\vec{w},b}\left(\vec{x}^{(i)}\right)\right)$$

cost

$$J(\vec{w}, b) = \frac{1}{m}\sum_{i=1}^{m}\left[L\left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right), y^{(i)}\right)\right]$$

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

We specifically chose this cost function instead of other function because there is a concept in statistics called maximum likelihood estimation which predicts best parameters for a cost function. Based on that algo we chose this cost function (more details on max likelihood not in the course)

Note: Gradient Descent formula for Logistic regression is same as linear regression. The only change is f(x) is now = sigmoid(z)

Overfitting, Underfitting and Generalization:

Underfit: Model with "High-Bias" which predicts values far from actual values

Overfit: Model which predicts exact values as actual values of a dataset but might fail for new example i.e. it has "high-variance" since it will predict totally different values for a different dataset.

Generalization: Model where bias is low, but new predictions might be closer to actual data.

How to resolve overfitting:

There are couple of different ways to tackle overfitting:

1. **Add large number of examples in dataset** - By adding more supervised examples, the curve will fit better and will have less variance for new dataset
2. **Eliminate Features** - If a dataset has 100 features for prediction, then select only limited number of features which you think will contribute to prediction.

This is called “Feature Selection”, although the disadvantage of this is that we can loose important information regarding prediction.

3. **Regularization**: Here instead of removing a feature completely from the equation, we instead reduce its weight to negligible values so that we can evaluate impact of a feature removal on predictions. This way we prevent any data loss.

Regularization:

In regularization, we try to reduce the weights of features such that it minimize the overfitting of data, hence we add a regularization term to cost function which actually decreases weights of features.
Since we don’t know which features to choose, hence initially we apply regularization to all features i.e. we decrease weight of all features by adding a regularization parameter.

Lambda here is called “Regularization Parameter”. If lambda is too low, then model will overfit since it won’t decrease the weights much whereas if lambda is too high then model will underfit since weights will be almost zero.

Hence we need to balance value of lambda such that it generalize the model fit.

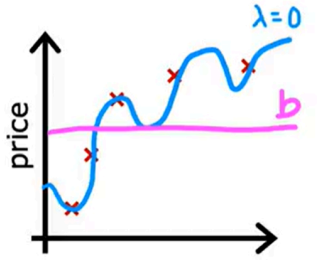
Regularization

$$\min_{\vec{w},b} J(\vec{w},b) = \min_{\vec{w},b} \left[\overbrace{\frac{1}{2m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2}^{\text{mean squared error}} + \overbrace{\frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2}^{\text{regularization term}} \right]$$

fit data

Keep w_j small

λ balances both goals



choose $\lambda = 10^{10}$

$$f_{\vec{w},b}(\vec{x}) = \underset{\approx 0}{w_1 x} + \underset{\approx 0}{w_2 x^2} + \underset{\approx 0}{w_3 x^3} + \underset{\approx 0}{w_4 x^4} + b$$

$$f(x) = b$$

Regularized Gradient Descent:

Since now we have updated our cost function, hence Gradient descent for linear regression now has a slightly different expression. Below we calculate derivative term dj/dw (gradient) for linear regression gradient descent:

How we get the derivative term (optional)

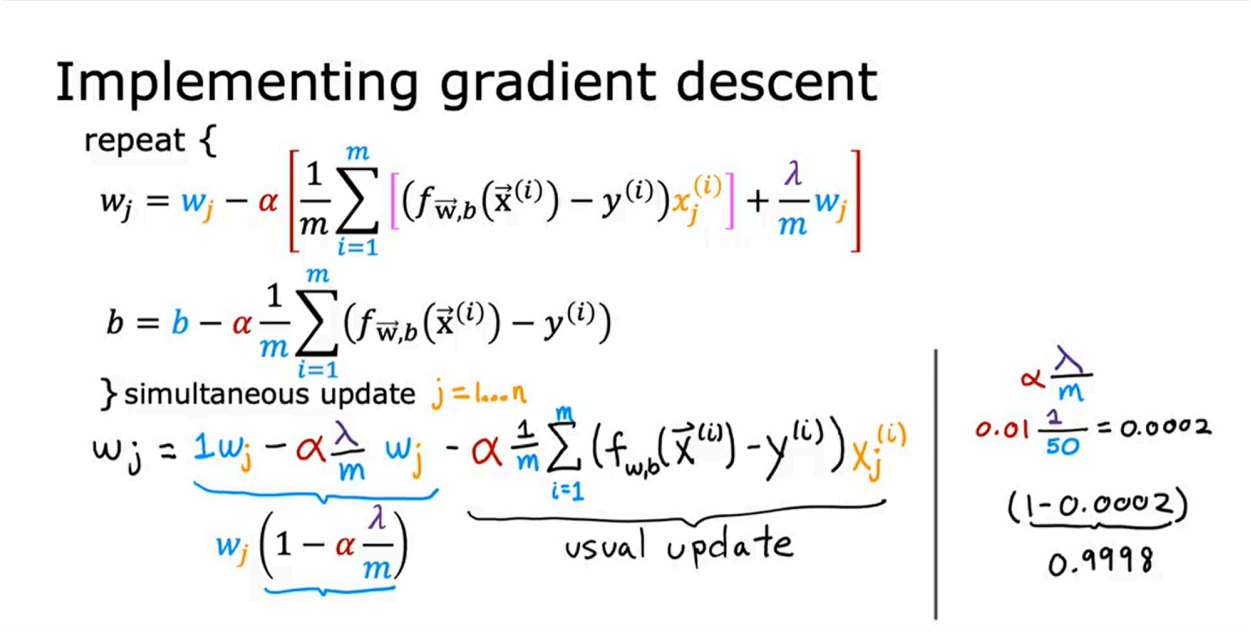
$$\frac{\partial}{\partial w_j} J(\vec{w},b) = \frac{\partial}{\partial w_j}\left[\frac{1}{2m}\sum_{i=1}^{m}\left(\underbrace{f(\vec{x}^{(i)})}_{\vec{w}\cdot\vec{x}^{(i)}+b} - y^{(i)}\right)^2 + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2\right]$$

$$= \frac{1}{2m}\sum_{i=1}^{m}\left[\left(\vec{w}\cdot\vec{x}^{(i)} + b - y^{(i)}\right) 2x_j^{(i)}\right] + \frac{\lambda}{2m} 2w_j \quad \text{No } \sum_{j=1}^{n}$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left[\left(\underbrace{\vec{w}\cdot\vec{x}^{(i)} + b}_{f(\vec{x})} - y^{(i)}\right) x_j^{(i)}\right] + \frac{\lambda}{m} w_j$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left[\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}\right] + \frac{\lambda}{m} w_j$$

Hence our overall gradient descent for linear regression looks like following:



Here we observe that when we implement gradient descent with regularization parameter then with each iteration the value of Wj (i.e. weight of feature j), reduces by (1 - (alpha*lambda)/m))

Logistic Regression Gradient Descent With Regularization:

Check overfitting_regularization jupyter notebook

Regularized logistic regression

$$J(\vec{w}, b) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) + (1-y^{(i)})\log\left(1-f_{\vec{w},b}(\vec{x}^{(i)})\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$$

$\min_{\vec{w},b}$

Gradient descent

Looks same as for linear regression!

repeat {

$$w_j = w_j - \alpha\frac{\partial}{\partial w_j}J(\vec{w}, b) \quad j = 1 \ldots n$$

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

logistic regression

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

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

don't have to regularize

}