Notes of Machine Learning Specialization

FlowerMouse

July 9, 2024

Overview

0.1 What you will learn?

- Build ML models with NumPy & scikit-learn, build & train supervised models for prediction & binary classification tasks (linear, logistic regression)
- Build & train a neural network with TensorFlow to perform multi-class classification, & build & use decision trees & tree ensemble methods
- Apply best practices for ML development & use unsupervised learning techniques for unsupervised learning including clustering & anomaly detection
- Build recommender systems with a collaborative filtering approach & a content-based deep learning method & build a deep reinforcement learning model

0.2 What is machine learning?

Machine learning is a field of artificial intelligence that uses statistical techniques to give computer systems the ability to ”learn” (i.e., progressively improve performance on a specific task) with data, without being explicitly programmed.

Types of machine learning

There are three main types of machine learning:

- Supervised learning
- Unsupervised learning
- Reinforcement learning

Supervised learning

Definition 0.2.1 ▶ Supervised Learning

Supervised Learning is a type of machine learning where the model is trained on a labeled dataset.

Unsupervised learning

Definition 0.2.2 ▶ Unsupervised Learning

Unsupervised Learning is a type of machine learning where the model is trained on an unlabeled dataset.

Reinforcement learning

Definition 0.2.3 ▶ Reinforcement Learning

Reinforcement Learning is a type of machine learning where the model learns to make decisions by interacting with an environment.

Contents

Overview **2**

0.1 What you will learn? . . . 2

0.2 What is machine learning? . . . 2

Part One Supervised Machine Learning: Regression and Classification **1**

1 Linear Regression with One Variable **2**

1.1 Model Representation . . . 2

2 Cost Function **3**

2.1 What is a cost function? . . . 3

2.2 Cost Function Intuition . . . 3

3 Gradient Descent **8**

3.1 Visualizing Gradient Descent . . . 8

3.2 Algorithm . . . 8

3.3 Derivatives . . . 10

4 Mutiple Features **12**

4.1 Notation . . . 12

4.2 Hypothesis Function . . . 12

4.3 Vectorization . . . 13

4.4 Gradient Descent for Multiple Features . . . 14

4.4.1 An alternative to the Gradient Descent . . . 15

4.4.2 implementation from scratch . . . 15

5 Tips for Gradient Descent **20**

5.1 Feature scaling . . . 20

5.2 Normalization . . . 21

5.3 Check the convergence . . . 23

5.4 Choose learning rate . . . 24
5.5 Polynomial regression . . . 25

6 Classification **27**
6.1 Linear Regression approach to Classification . . . 27
6.2 Logistic Regression . . . 28
6.2.1 Decision Boundary . . . 29
6.3 Cost Function . . . 31
6.4 Gradient Descent . . . 32

7 The problem of overfitting **34**
7.1 Overfitting . . . 34
7.2 Addressing overfitting . . . 35
7.3 Regularization . . . 36
7.3.1 regularization for linear regression . . . 36
7.3.2 regularization for logistic classification . . . 37

Part Two Advanced Machine Learning Algorithms **38**

8 Neural Networks Introduction **39**
8.1 Neural networks intuition . . . 39
8.1.1 Neuron and the brain . . . 39
8.1.2 Demand prediction . . . 39
8.2 Neural networks model . . . 41
8.3 Implementation in TensorFlow . . . 42
8.4 Building a neural network in TensorFlow . . . 44
8.5 Forward propagation . . . 45

9 Neural Networks Training **46**
9.1 Training . . . 46
9.2 Details . . . 46
9.3 Activation functions . . . 48
9.4 Multiclass Classification . . . 49
9.5 Multi-label Classification . . . 52
9.6 Adam Optimizer . . . 53
9.7 Additional layer types . . . 54
9.8 Computation graph . . . 55

10 Advice for applying machine learning **59**
10.1 Evaluating a model 59
10.2 Bias and variance 64
10.3 Baseline level of performance 66
10.4 Learning curves 67
10.5 Machine learning develop process 71
10.6 Skewed datasets 76

11 Decision Trees **80**
11.1 Introduction 80
11.2 Decision tree learning 83
11.3 Tree ensembles 88
11.3.1 XGBoost 91

Part Three Unsupervised Learning, Recommender Systems and Reinforcement Learning **93**

Index **94**

Part I

Supervised Machine Learning: Regression and Classification

1 Linear Regression with One Variable

1.1 Model Representation

Notation

- m = Number of training examples
- x = ”input” variable / features
- y = ”output” variable / ”target” variable
- (x, y) = one training example
- $(x^{(i)}, y^{(i)}) = i^{th}$ training example
- f = hypothesis function
- $\hat{y}$ = predicted value

Hypothesis Function (One Variable)

$$f_{w,b}(x) = w \cdot x + b \tag{1.1}$$

2 Cost Function

2.1 What is a cost function?

Definition 2.1.1 ▶ Cost Function

Cost Function is a function that measures the performance of a machine learning model.

$$Squared\ error\ cost\ function: \sum_{i=1}^{m} \frac{1}{2m} \left(\hat{y}^{(i)} - y^{(i)}\right)^2 \tag{2.1}$$

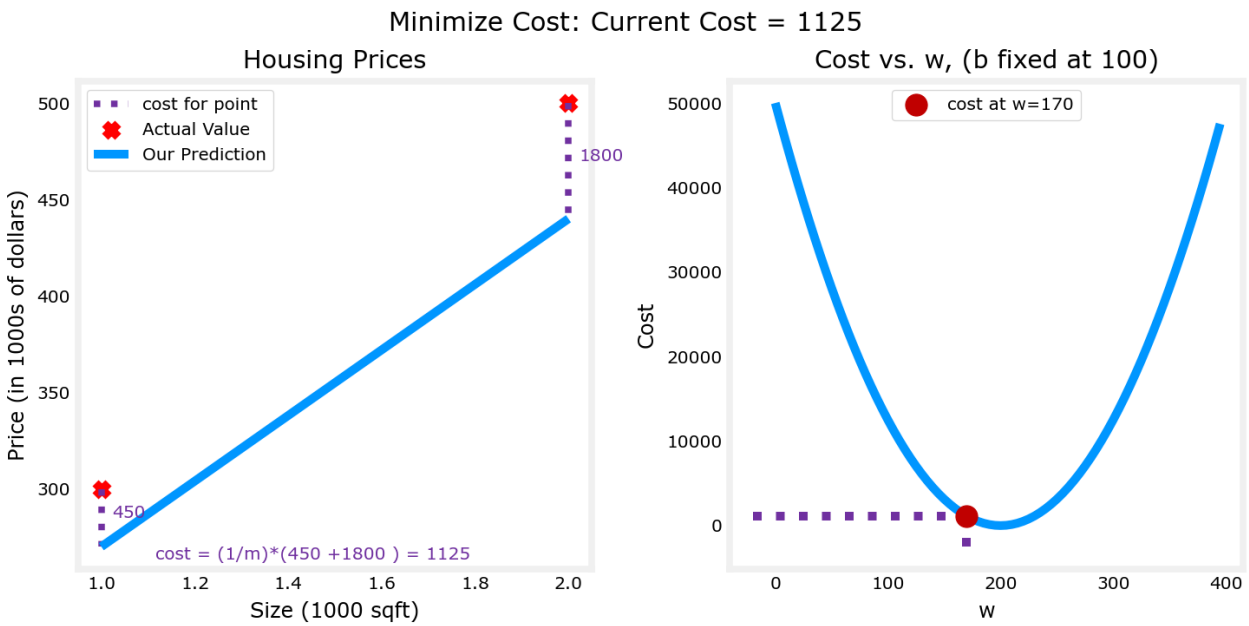
explaination the $\hat{y}^{(i)} - y^{(i)}$shows the distance of the predicted value from the actual value.

2.2 Cost Function Intuition

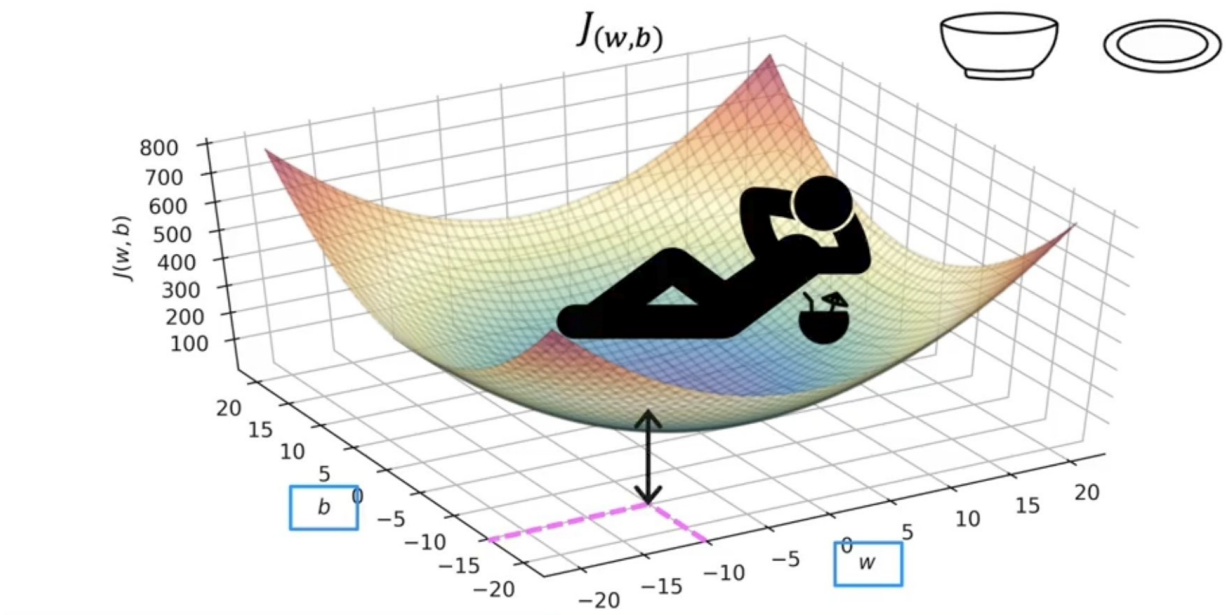
$J(w)$ with only one parameter w

Code Snippet 2.2.1 ▶ Cost Function

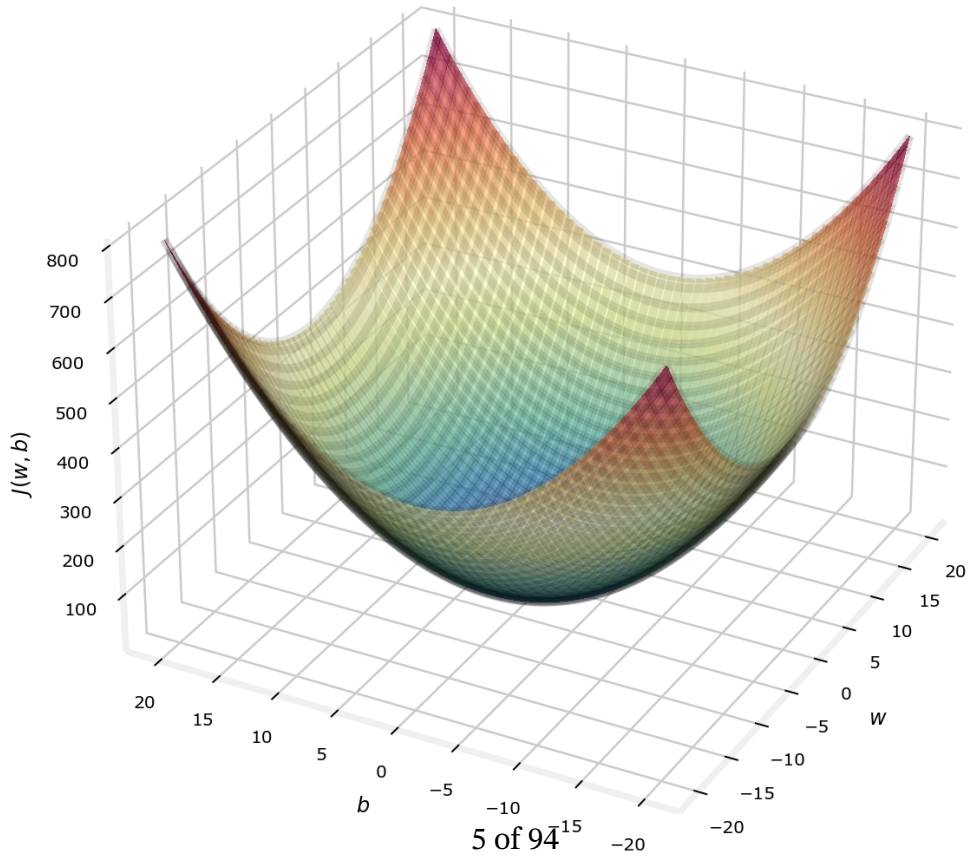
```
x_train = np.array([1.0, 2.0])           #(size in 1000 square feet)
y_train = np.array([300.0, 500.0])       #(price in 1000s of dollars)
plt_intuition(x_train,y_train)
```


Minimize Cost: Current Cost = 1125
Housing Prices
cost for point
Actual Value
Our Prediction
1800
450
cost = (1/m)*(450 +1800) = 1125
Price (in 1000s of dollars)
Size (1000 sqft)
Cost vs. w, (b fixed at 100)
cost at w=170
Cost
w

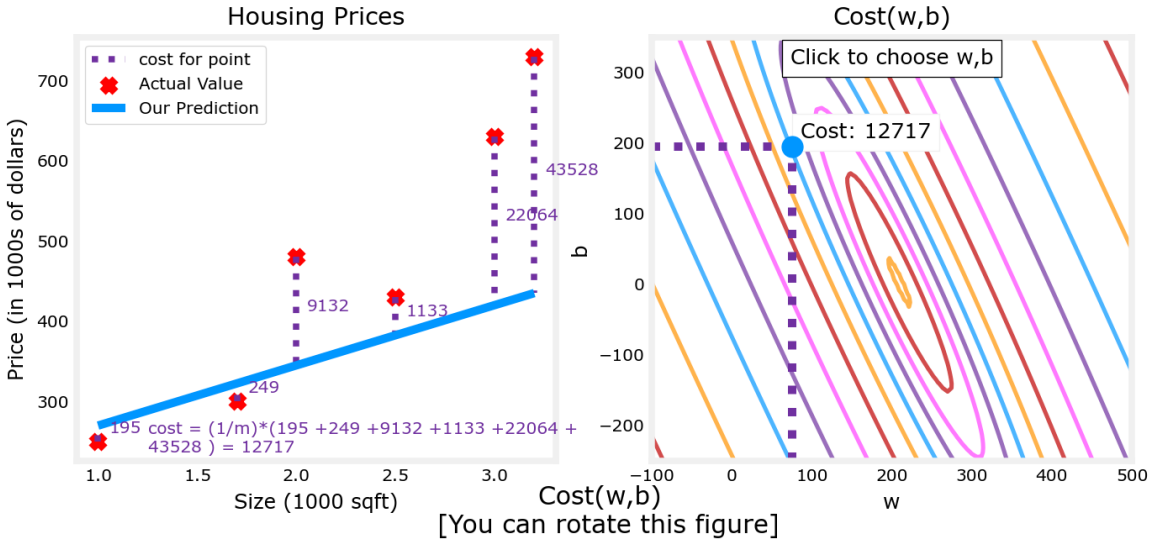
$J(w, b)$ with two parameters w and b

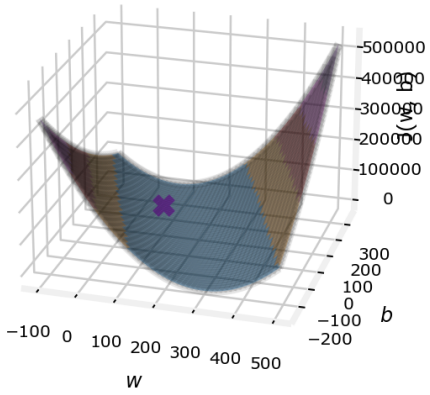


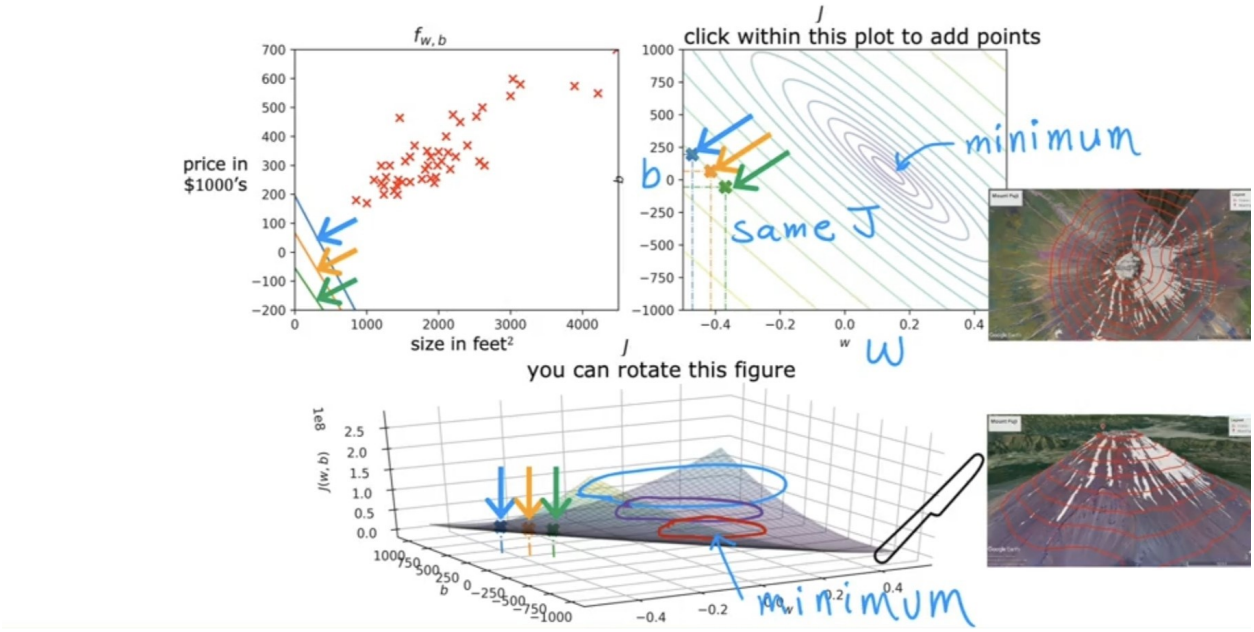
$J(w, b)$
[You can rotate this figure]



explaination The cost function $J(w, b)$ always has the shape of a "soup bowl".



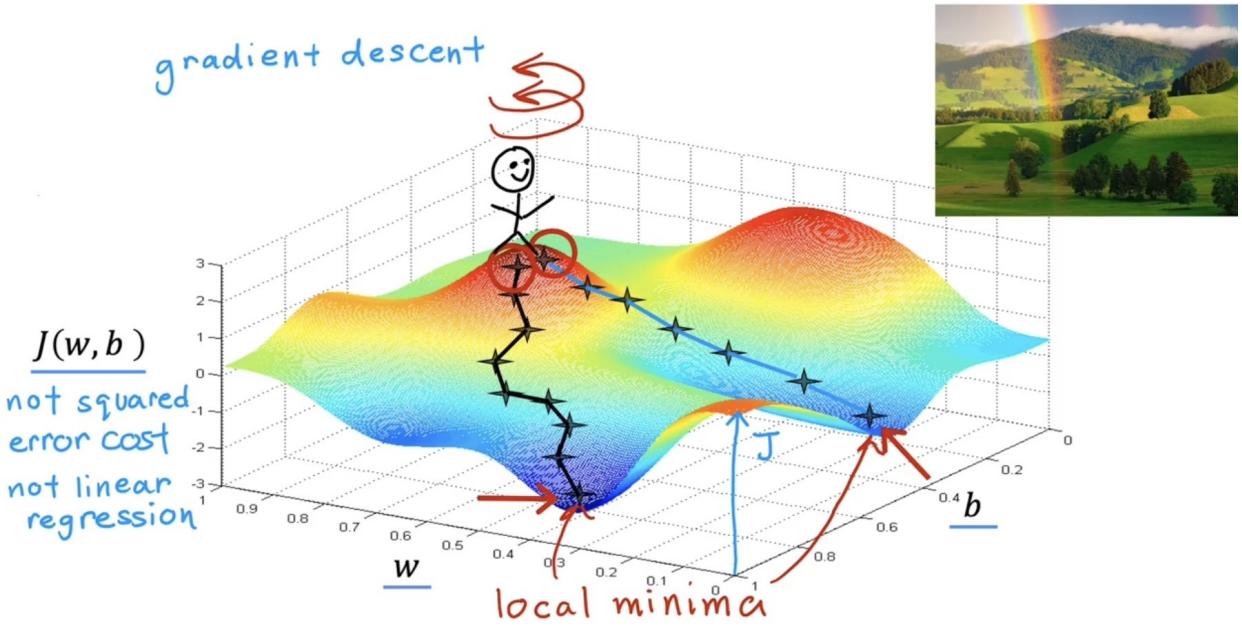




explaination The 3D plot can be transformed into a contour plot.

3 Gradient Descent

3.1 Visualizing Gradient Descent



Terminology: local minimas, global minimas, saddle points

3.2 Algorithm

Theorem 3.2.1 ▶ Gradient Descent Algorithm

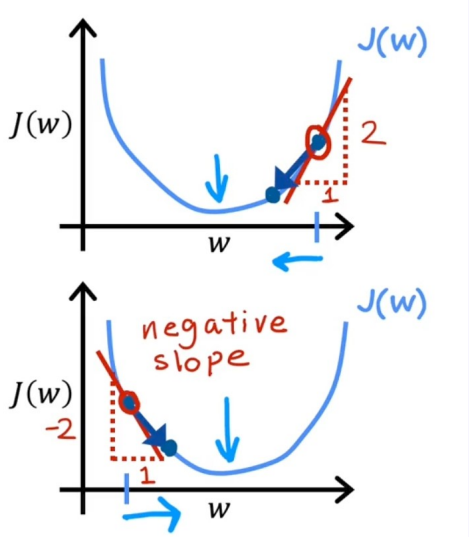
repeat until convergence

$$w = w - \alpha \cdot \frac{\partial}{\partial w} J(w, b) \tag{3.1}$$

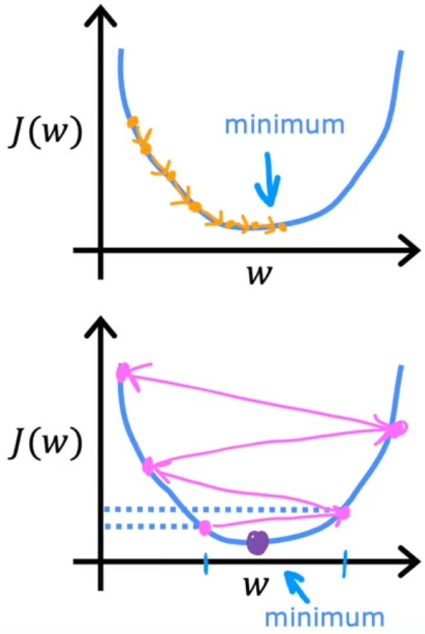
$$b = b - \alpha \cdot \frac{\partial}{\partial b} J(w, b) \tag{3.2}$$

- note that = represents assignment, not equality
- α is the learning rate
- It is important to **simutaniously** update w and b

when $w > w_{min}$, the derivative is positive, so w will decrease
when $w < w_{min}$, the derivative is negative, so w will increase



Learning Rate

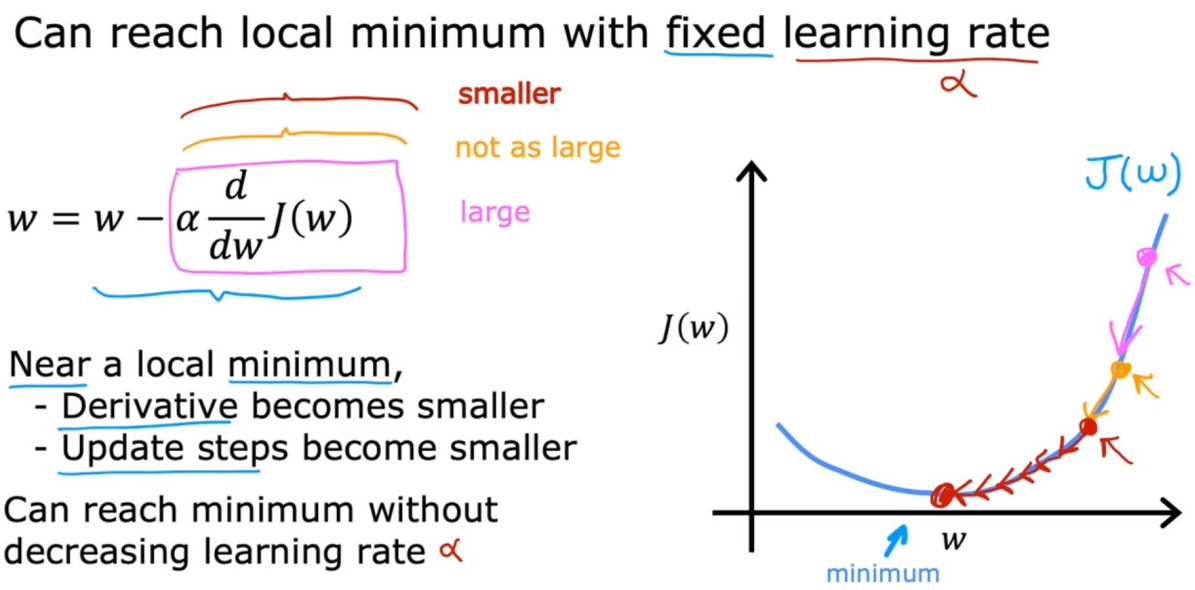


1. If the learning rate is too small, the algorithm will take a long time to converge
2. If the learning rate is too large, the algorithm may overshoot the minimum or even diverge.

Example 3.2.2 ▸ local minima

if the w eactly at the local minima, the derivative is 0,the w remains the same according to the equation so the algorithm will stop.

why is it can reach local minima with fixed learning rate?



When near a local minima, the derivative become smaller, update will be smaller, so the algorithm will converge to the local minima without decreasing learning rate.

3.3 Derivatives

Theorem 3.3.1 ▸ imporved Gradient Descent Algorithm

$$w = w - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} (w \cdot x^{(i)} + b - y^{(i)}) \cdot x^{(i)} \tag{3.3}$$

$$b = b - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} (w \cdot x^{(i)} + b - y^{(i)}) \tag{3.4}$$

Proof.

$$\begin{aligned} J(w,b) &= \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2 \\ &= \frac{1}{2m}\sum_{i=1}^{m}(w \cdot x^{(i)} + b - y^{(i)})^2 \end{aligned}$$

$$\begin{aligned} \frac{\partial}{\partial w}J(w,b) &= \frac{\partial}{\partial w}\frac{1}{2m}\sum_{i=1}^{m}(w \cdot x^{(i)} + b - y^{(i)})^2 \\ &= \frac{1}{2m}\sum_{i=1}^{m}2 \cdot (w \cdot x^{(i)} + b - y^{(i)}) \cdot x^{(i)} \\ &= \frac{1}{m}\sum_{i=1}^{m}(w \cdot x^{(i)} + b - y^{(i)}) \cdot x^{(i)} \end{aligned}$$

$$\begin{aligned} \frac{\partial}{\partial b}J(w,b) &= \frac{\partial}{\partial b}\frac{1}{2m}\sum_{i=1}^{m}(w \cdot x^{(i)} + b - y^{(i)})^2 \\ &= \frac{1}{2m}\sum_{i=1}^{m}2 \cdot (w \cdot x^{(i)} + b - y^{(i)}) \\ &= \frac{1}{m}\sum_{i=1}^{m}(w \cdot x^{(i)} + b - y^{(i)}) \end{aligned}$$

□

note that the equared error cost function is **convex**, so there is no local minima, only global minima.
so the algorithm will always converge to the global minima.

Definition 3.3.2 ▸ Batch Gradient Descent

the “batch” in the name means that the algorithm uses all the training examples to update the parameters.
some other algorithms use only a subset of the training examples.

4 Mutiple Features

4.1 Notation

Example 4.1.1 ▸ Notation for mutiple features

$$
\begin{aligned}
n &= \text{number of features} \\
x^{(i)} &= \text{input (features) of the } i^{th} \text{ training example} \\
x_j^{(i)} &= \text{value of feature } j \text{ in the } i^{th} \text{ training example} \\
m &= \text{number of training examples}
\end{aligned}
$$

$$
\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}
$$

$$
\vec{x}^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix}
$$

4.2 Hypothesis Function

Definition 4.2.1 ▸ Hypothesis Function

Mutiple linear regression

$$
f_{\vec{w},b} = \vec{w} \cdot \vec{x} + b = \vec{w}^T \vec{x} + b \tag{4.1}
$$

$$\vec{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \quad \vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

the $\vec{w}$ and $\vec{x}$ are both column vectors, but b is a number.

4.3 Vectorization

The vectorized implementation is much more efficient than the for-loop implementation.

For-loop implementation for a sigle training example:

```
# n is the number of features
for i in range(n):
    f += w[i] * x[i]
f += b
```

Vectorized implementation for a sigle training example:

```
f = np.dot(w, x) + b
f = w @ x + b
# f = w * x + b  this is wrong, numpy will broadcast
```

np.dot can be used to calculate the dot product of two arrays if they are 1-D arrays.
and np.dot can also be used to calculate the matrix multiplication if they are 2-D arrays.

4.4 Gradient Descent for Multiple Features

Definition 4.4.1 ▶ Cost Function for Multiple Features

The equation for the cost function with multiple variables $J(\mathbf{w}, b)$ is:

$$J(\mathbf{w}, b) = \frac{1}{2m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})^2 \tag{4.2}$$

where:

$$f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = \mathbf{w} \cdot \mathbf{x}^{(i)} + b \tag{4.3}$$

In contrast to previous labs, $\mathbf{w}$ and $\mathbf{x}^{(i)}$ are vectors rather than scalars supporting multiple features.

Theorem 4.4.2 ▶ Gradient Descent for Multiple Features

Gradient descent for multiple variables:
reapeat until convergence:

$$w_j := w_j - \alpha \frac{\partial J(\mathbf{w}, b)}{\partial w_j} \qquad for\ j = 0, \cdots, n-1 \tag{4.4}$$

$$b := b - \alpha \frac{\partial J(\mathbf{w}, b)}{\partial b} \tag{4.5}$$

where, n is the number of features, parameters w_j, b, are updated simultaneously and where

$$\frac{\partial J(\mathbf{w}, b)}{\partial w_j} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)} \tag{6}$$

$$\frac{\partial J(\mathbf{w}, b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) \tag{7}$$

- m is the number of training examples in the data set
- $f_{\mathbf{w},b}(\mathbf{x}^{(i)})$ is the model's prediction, while $y^{(i)}$ is the target value

4.4.1 An alternative to the Gradient Descent

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 ($> 10,000$)

> - 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

4.4.2 implementation from scratch

file name: multi-lin-regression.py

```
import numpy as np
import matplotlib as plt
from copy import deepcopy
from sklearn.datasets import fetch_openml
from sklearn.linear_model import LinearRegression

def predict(x, w, b):
    """
    single predict using linear regression

    Args:
      x (ndarray): Shape (n,) example with multiple features
      w (ndarray): Shape (n,) model parameters
      b (scalar):  model parameter

    Returns:
      p (scalar):  prediction
```

```
    """
    p = np.dot(x, w) + b
    return p

def compute_cost(X, y, w, b):
    """
    Args:
      X (ndarray (m,n)): Data, m examples with n features
      y (ndarray (m,)) : target values
      w (ndarray (n,)) : model parameters
      b (scalar)       : model parameter

    Returns:
      cost (scalar): cost
    """
    m = X.shape[0]
    cost = 0.0
    for i in range(m):
        f_wb_i = np.dot(X[i], w) + b           #(n,)(n,) = scalar (see np.dot)
         cost = cost + (f_wb_i - y[i])**2       #scalar
    cost = cost / (2 * m)                       #scalar
    return cost

def compute_gradient(X, y, w, b):
    """
    Computes the gradient for linear regression
    Args:
      X (ndarray (m,n)): Data, m examples with n features
      y (ndarray (m,)) : target values
      w (ndarray (n,)) : model parameters
      b (scalar)       : model parameter

    Returns:
      dj_dw (ndarray (n,)): The gradient of the cost w.r.t. the parameters
       ↪  w.
```

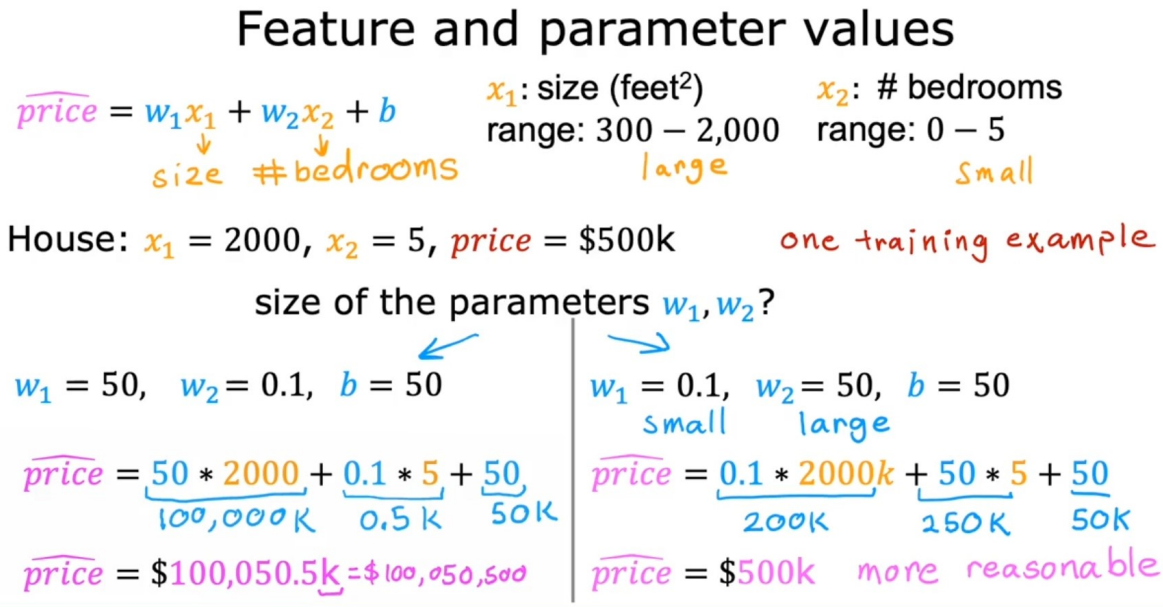
```
      dj_db (scalar):       The gradient of the cost w.r.t. the parameter b.
    """
    m, n = X.shape           # (number of examples, number of features)
    dj_dw = np.zeros((n,))
    dj_db = 0.

    for i in range(m):
        err = (np.dot(X[i], w) + b) - y[i]
        for j in range(n):
            dj_dw[j] = dj_dw[j] + err * X[i, j]
        dj_db = dj_db + err
    dj_dw = dj_dw / m
    dj_db = dj_db / m

    return dj_db, dj_dw

def gradient_descent(X, y, w_in, b_in, cost_function, gradient_function,
 ↪  alpha, num_iters):
    """
    Performs batch gradient descent to learn theta. Updates theta by taking
    num_iters gradient steps with learning rate alpha

    Args:
      X (ndarray (m,n))   : Data, m examples with n features
      y (ndarray (m,))    : target values
      w_in (ndarray (n,)) : initial model parameters
      b_in (scalar)       : initial model parameter
      cost_function       : function to compute cost
      gradient_function   : function to compute the gradient
      alpha (float)       : Learning rate
      num_iters (int)     : number of iterations to run gradient descent

    Returns:
      w (ndarray (n,)) : Updated values of parameters
      b (scalar)       : Updated value of parameter
```

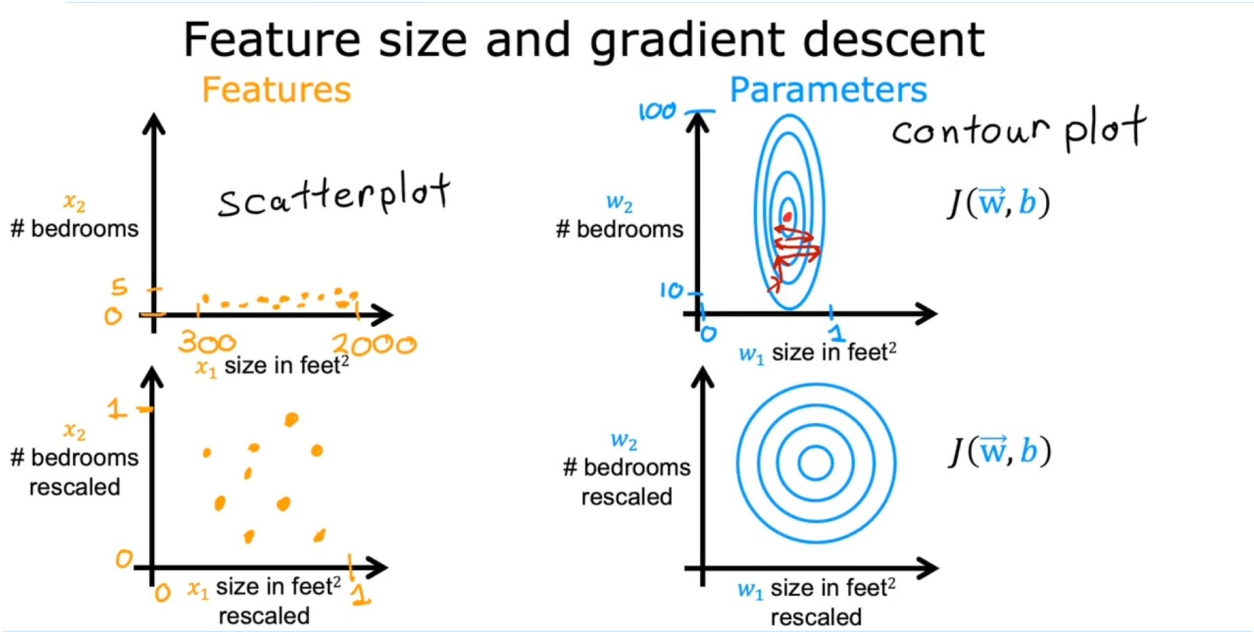
```
      """
    w = deepcopy(w_in)  # avoid modifying global w within function
    b = b_in

    for i in range(num_iters):

        # Calculate the gradient and update the parameters
        dj_db,dj_dw = gradient_function(X, y, w, b)

        # Update Parameters using w, b, alpha and gradient
        w = w - alpha * dj_dw
        b = b - alpha * dj_db

    return w, b     # return final w, b

def implement(X, y):
    # initialize parameters
    initial_w = np.ones_like(X[0])
    initial_b = 1.
    # some gradient descent settings
    iterations = 10000
    alpha = 5.0e-2
    # run gradient descent
    w_final, b_final = gradient_descent(X, y, initial_w, initial_b,
                                        compute_cost, compute_gradient,
                                            alpha, iterations)
    print(f"b, w found by gradient descent: {b_final:0.2f},{w_final} ")
    m, _ = X.shape
    for i in range(m):
        print(f"prediction: {predict(X[i], w_final, b_final)}, target value:
          ↪ {y[i]}")

def normalize(X):
    """
    z-score normalization
```

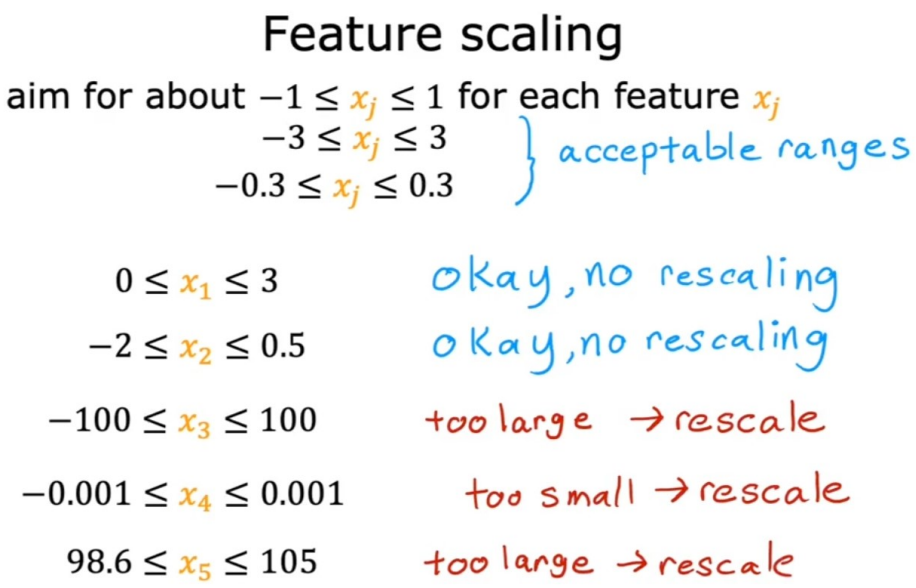
```
    """
    # avoid division by zero
    eps = 1e-8
    for i in range(X.shape[1]):
        X[:, i] = (X[:, i] - np.mean(X[:, i])) / (np.std(X[:, i]) + eps)

def main():
    boston = fetch_openml(name='boston', version=1)
    X = boston.data.to_numpy(dtype=np.float32)[:10, :]
    normalize(X)
    print(X)
    y = boston.target.to_numpy(dtype=np.float32)[:10]
    implement(X, y)

    # compare with sklearn
    model = LinearRegression()
    model.fit(X, y)
    print(f"b, w found by sklearn: {model.intercept_},{model.coef_} ")
    m, _ = X.shape
    for i in range(m):
        print(f"prediction: {model.predict([X[i]])}, target value: {y[i]}")

main()
```

Tips for Gradient Descent

5.1 Feature scaling







5.2 Normalization

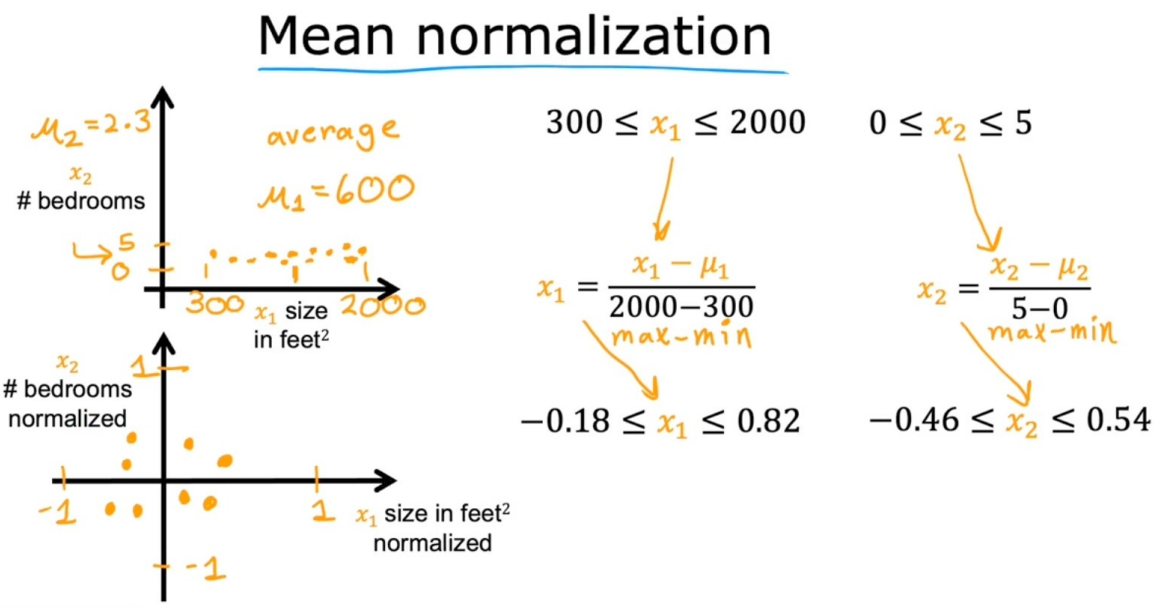
Mean normalization

Theorem 5.2.1 ▶ mean normalization

for each feature x_i

$$x_i = \frac{x_i - \mu_i}{\max - \min} \tag{5.1}$$

μ_i is the average value of x_i



Z-score normalization

Theorem 5.2.2 ▶ z-score normalization

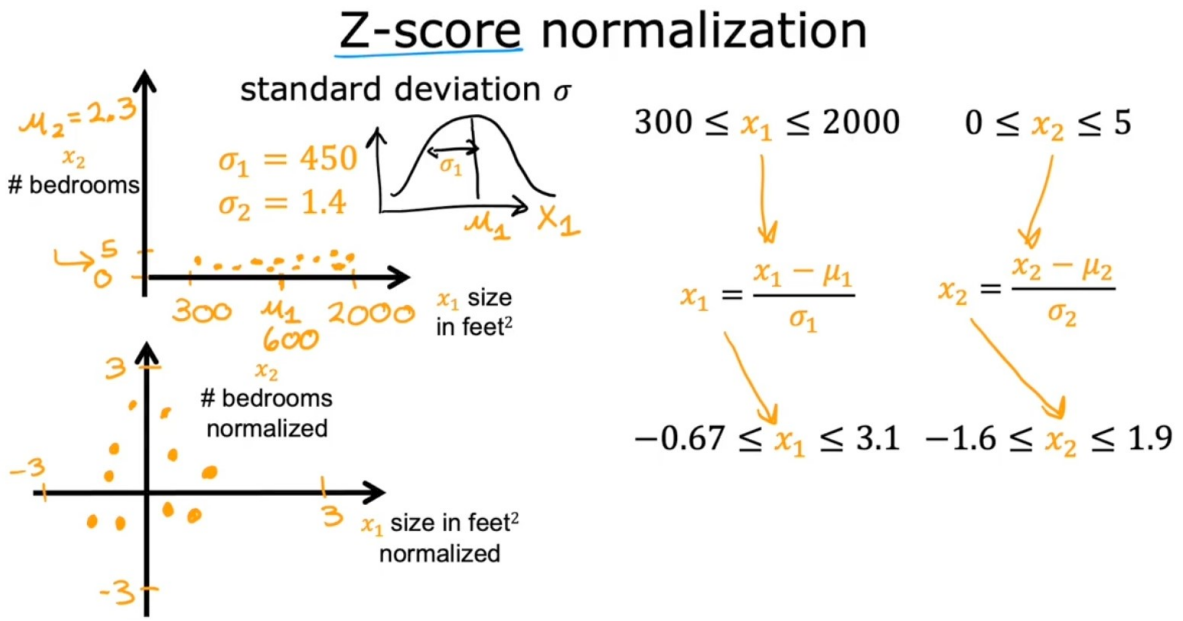
for each feature x_i

$$x_i = \frac{x_i - \mu_i}{\sigma_i} \tag{5.2}$$

μ_i is the average value of x_i, σ_i is the standard deviation of x_i

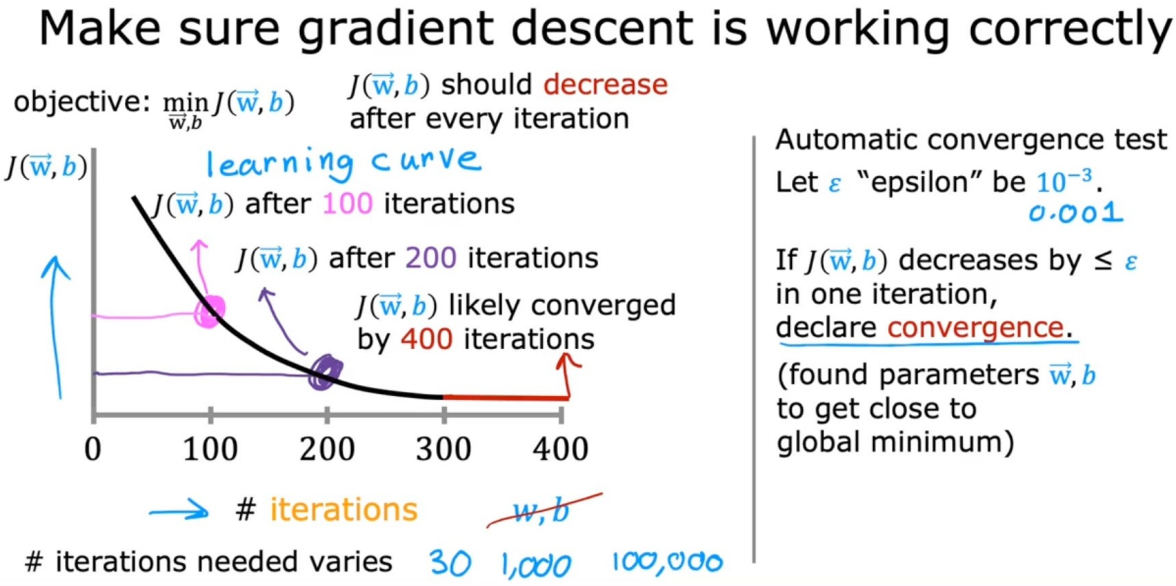
$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)}$$

$$\sigma_i = \sqrt{\frac{1}{m}\sum_{j=1}^{m}(x_i^{(j)} - \mu_i)^2}$$

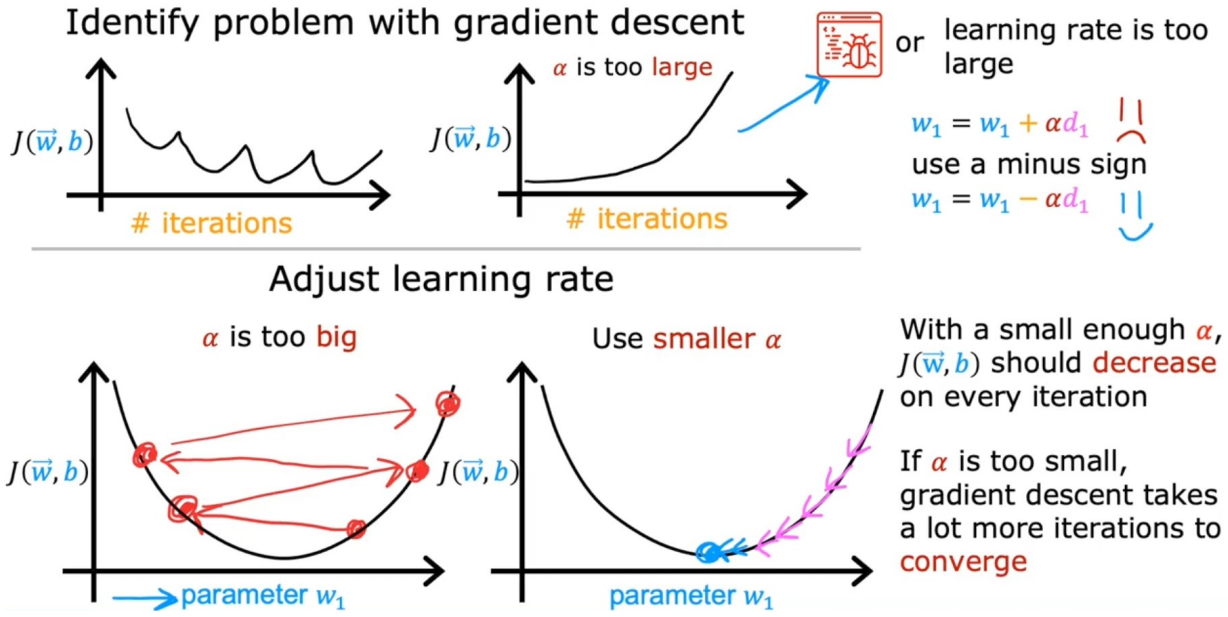


5.3 Check the convergence

the cost versus iterations gragh A plot of cost versus iterations is a useful measure of progress in gradient descent. Cost should always decrease in successful runs. The change in cost is so rapid initially, it is useful to plot the initial decent on a different scale than the final descent. In the plots below, note the scale of cost on the axes and the iteration step.



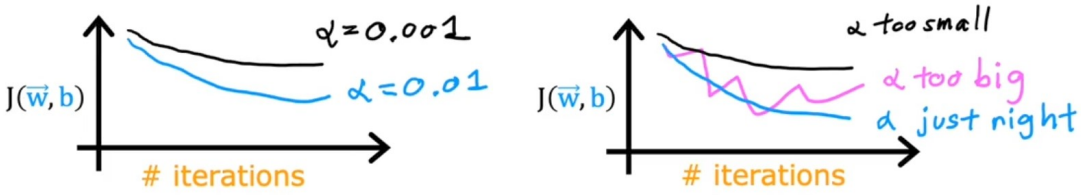
5.4 Choose learning rate



Values of α to try:

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

3X ≈3X 3X ≈3X 3X ≈3X



5.5 Polynomial regression

Definition 5.5.1 ▶ Polynomial regression

Polynomial regression is a form of regression analysis in which the relationship between the independent variable x and the dependent variable y is modelled as an nth degree polynomial in x.

Example 5.5.2 ▶ Polynomial regression

consider the following data: assume $f_{(w,b)} = w \cdot x^2 + b$

By looking at the data, we can see that the data is not linear, so we can use polynomial regression to fit the data.

So we choose a new feature x^2 and apply linear regression to the new feature.(feature engnieering).

Feature engineering

1. first, we need to normalize the features.
2. Usually, it is hard to know which feature to use, so we can try different features and see which one fits the data best.
3. for example, we can try x^2, x^3, x^4, x^5 and so on.
4. so the hypothesis function will be $f_{(w,b)} = w_1 \cdot x + w_2 \cdot x^2 + w_3 \cdot x^2 + b$
5. apply linear regression to the new feature.
6. Eventally, we can get the funtions looks like this for this example: $f_{(w,b)} = 0.08x + 0.54x^2 + 0.03x^3 + 0.0106$
7. Let's review this idea:
   (a) Intially, the features were re-scaled so they are comparable to each other
   (b) less weight value implies less important/correct feature, and in extreme, when the weight becomes zero or very close to zero, the associated feature useful in fitting the model to the data.

(c) above, after fitting, the weight associated with the x^2 feature is much larger than the weights for x or x^3 as it is the most useful in fitting the data.

the essence of polynomial regression is to use linear regression to fit the data, but with the help of feature engineering, we can fit the data better.

even when we create new features, we still use linear regression to fit the data.

6 Classification

Definition 6.0.1 ▶ Classification

Classification is a process of categorizing a given set of data into classes. It can be performed on both structured or unstructured data. The process starts with predicting the class of given data points. The classes are often referred to as target, label or categories.

Definition 6.0.2 ▶ Binary classification

Binary or binomial classification is the task of classifying the elements of a given set into two groups on the basis of a classification rule. In binary classification, the output is a binary value, such as yes/no, 0/1, true/false, etc.

Usually, the two classes/categories are represented by 0 and 1. we call the 0 as the negative class and the 1 as the positive class.

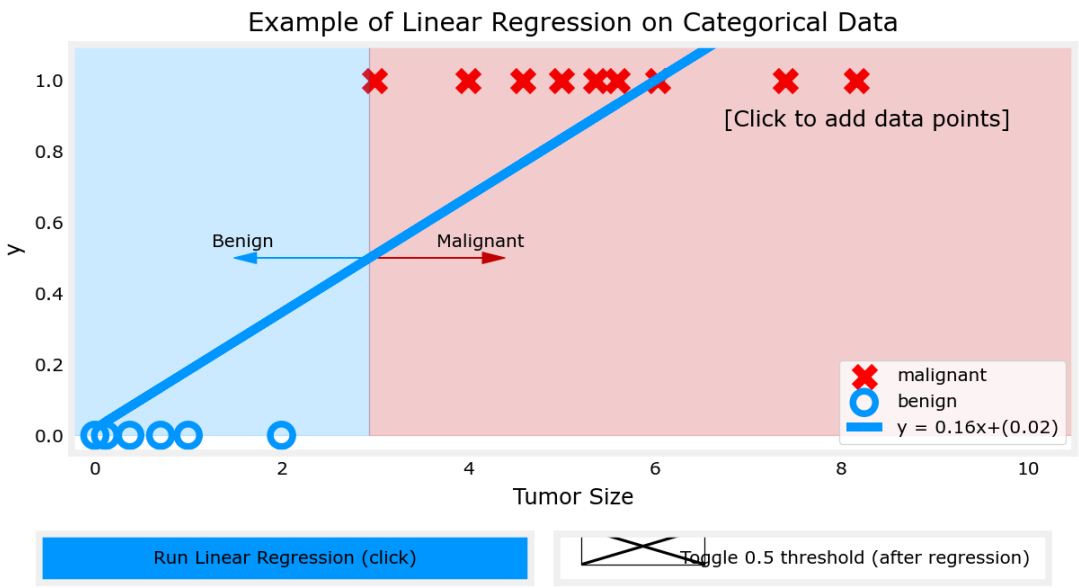
6.1 Linear Regression approach to Classification

we can use linear regression for classification problems, but it is not a good idea. The problem with linear regression is that it tries to predict the output as a continuous value. In the case of binary classification, the output is a binary value, 0 or 1.

Example 6.1.1 ▶ Linear Regression for Binary Classification

Let's say we have a binary classification problem with two classes, 0 and 1. We can use linear regression to predict the output. The output of linear regression can be any real number, which can be greater than 1 or less than 0. We can set a threshold value, say 0.5. If the output is greater than 0.5, we can classify it as 1, and if it is less than 0.5, we can classify it as 0.

Take the example of predicting a tumor as malignant or benign.



In the above figure, if we add more malignant tumors with a large size, the line will shift towards the malignant side. This will cause the benign tumors to be classified as malignant. However, the benign tumors are still benign, but they are classified as malignant because of the shift in the line.

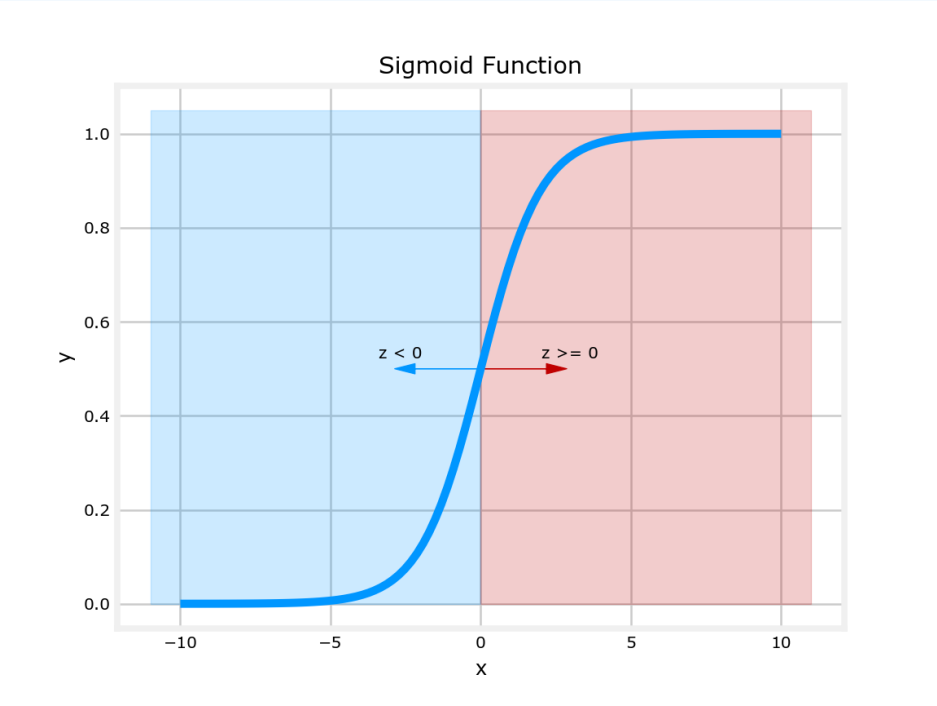
6.2 Logistic Regression

Definition 6.2.1 ▶ Logistic Regression

Logistic regression is a statistical model that in its basic form uses a logistic function to model a binary dependent variable.

sigmoid function/logistic function ：

$$g(z) = \frac{1}{1+e^{-z}} \qquad 0 < g(z) < 1 \tag{6.1}$$



Theorem 6.2.2 ▶ Logistic Regression

we can transform the output of linear regression using the sigmoid function.

$$
\begin{aligned}
f_{(\mathbf{w},b)}(\mathbf{x}) &= g(z) \\
z &= \mathbf{w}^T\mathbf{x} + b \\
g(z) &= \frac{1}{1+e^{-z}} \\
&\Downarrow \\
f_{(\mathbf{w},b)}(\mathbf{x}) &= g(\mathbf{w}^T\mathbf{x}+b) = \frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}
\end{aligned}
$$

Interpretation of the output The output of logistic regression is the probability that the given input point belongs to the positive class.(class 1)

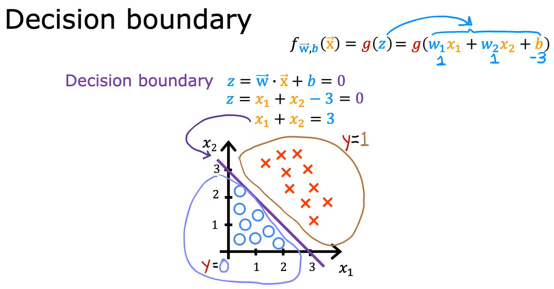
$$f_{(\mathbf{w},b)}(\mathbf{x}) = P(y=1|\mathbf{x};\mathbf{w},b) \tag{6.2}$$

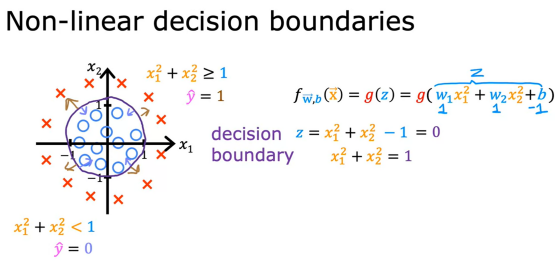
6.2.1 Decision Boundary

How to classify? when the output of logistic regression is greater than 0.5, we classify it as 1, and when it is less than 0.5, we classify it as 0.

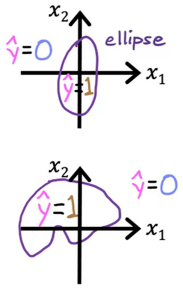
By looking at the sigmoid function, we can see that the output of logistic regression is greater than 0.5 when $z > 0$ and less than 0.5 when $z < 0$. Which is to say when $\mathbf{w}^T\mathbf{x} + b > 0$, we classify it as 1, and when $\mathbf{w}^T\mathbf{x} + b < 0$, we classify it as 0.

The decision boundary is the line that separates the positive class from the negative class. It is the line where $z = 0$. In the function diagram, we can draw the image of the function $z = 0$. For example, if we have two features x_1 and x_2, the decision boundary is the line where $w_1x_1 + w_2x_2 + b = 0$. and the z can be also other comlex functions. Such as $z = w_1x_1^2 + w_2x_2^2 + b$, which is a circle.





Non-linear decision boundaries



$$f_{\vec{w},b}(\vec{x}) = g(z) = g(w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_1x_2 + w_5x_2^2 + w_6x_1^3 + \cdots + b)$$

6.3 Cost Function

why it is unreasonable to use the cost function of linear regression?

the squared error cost function of linear regression is not suitable for logistic regression. because when we use it, the cost function of logistic regression will be non-convex. which means it will have many local minimums, and gradient descent may not converge to the global minimum.

Definition 6.3.1 ▸ Logistic loss function

The cost function of logistic regression is defined as:

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

the simplified version is:

$$L(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)}) = -y^{(i)} \log(f_{\mathbf{w},b}(\mathbf{x}^{(i)})) - (1 - y^{(i)}) \log(1 - f_{\mathbf{w},b}(\mathbf{x}^{(i)})) \tag{6.3}$$

so the loss function is convex.

Definition 6.3.2 ▸ Logistic cost function

$$J(\mathbf{w}, b) = \frac{1}{m} \sum_{i=1}^{m} L(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)}) \tag{6.4}$$

$$J(\mathbf{w}, b) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right) \right] \tag{6.5}$$

the cost function of logistic regression is derived from the principle of maximum likelihood estimation.

6.4 Gradient Descent

Theorem 6.4.1 ▶ Gradient Descent

$$J(\mathbf{w}, b) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right) + \left(1-y^{(i)}\right)\log\left(1-f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right)\right] \tag{6.6}$$

The gradient descent algorithm is used to minimize the cost function. The update rule for the parameters is:

$$\begin{aligned} &repeat\ \{ \\ &\quad w_j := w_j - \alpha\frac{\partial}{\partial \mathbf{w_j}}J(\mathbf{w}, b) \\ &\quad b := b - \alpha\frac{\partial}{\partial b}J(\mathbf{w}, b) \\ &\}\quad simultaneously\ update \end{aligned}$$

where α is the learning rate. final version:

$$\begin{aligned} &repeat\ \{ \\ &\quad w_j := w_j - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}\right)x_j^{(i)} \qquad (6.7) \\ &\quad b := b - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}\right) \qquad (6.8) \\ &\}\quad simultaneously\ update \end{aligned}$$

the math derivation is below

$\frac{\partial}{\partial w_j} J(w,b)$ set $f_{w,b}(x^{(i)}) = g(z) = a$

$= \frac{\partial}{\partial w_j} -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log a + (1-y^{(i)}) \log (1-a) \right]$

$= -\frac{1}{m} \sum_{i=1}^{m} \left[\log(1-a) + y^{(i)} \log \frac{a}{1-a} \right]$

$a = \frac{1}{1+e^{-z}}$

$= -\frac{1}{m} \sum_{i=1}^{m} \left[\log \left(\frac{1}{e^z+1} \right) + y^{(i)} \log e^z \right]$

$= -\frac{1}{m} \sum_{i=1}^{m} \left[(e^z+1) \frac{-e^z}{(e^z+1)^2} \cdot \frac{\partial z}{\partial w_j} + y^{(i)} \frac{\partial z}{\partial w_j} \right]$

$= \sum_{i=1}^{m} \frac{1}{m} \left[\frac{1}{1+e^{-z}} \cdot \frac{\partial z}{\partial w_j} + y^{(i)} \frac{\partial z}{\partial w_j} \right]$

$z = w_1 x_1^{(i)} + w_2 x_2^{(i)} + \cdots + w_n x_n^{(i)}$

$\frac{\partial z}{\partial w_j} = x_j^{(i)}$

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

The same goes to

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

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

7

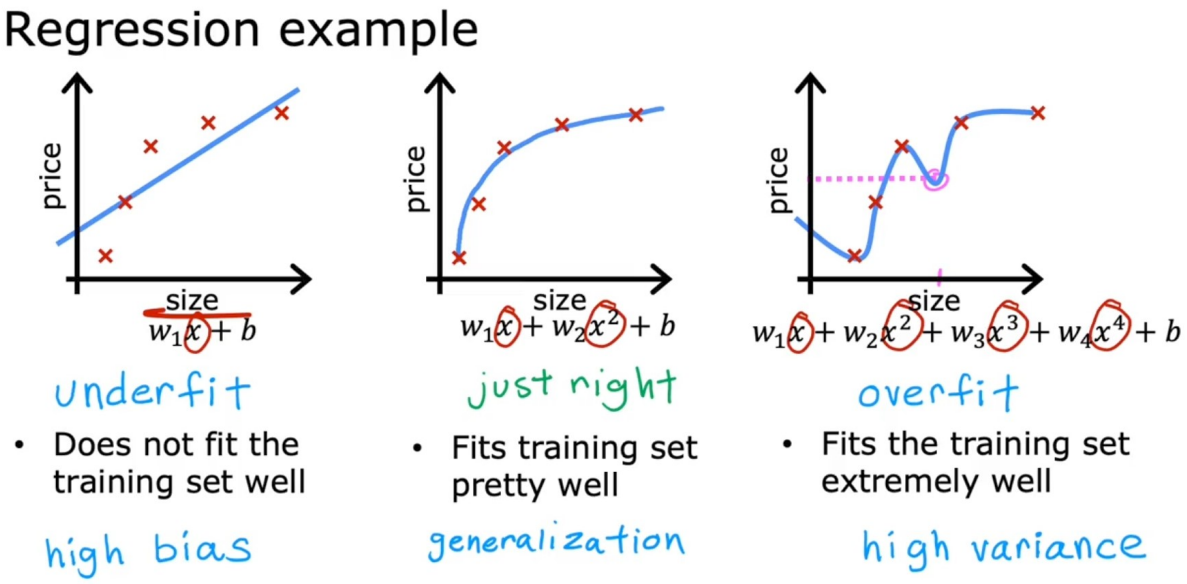
The problem of overfitting

7.1 Overfitting

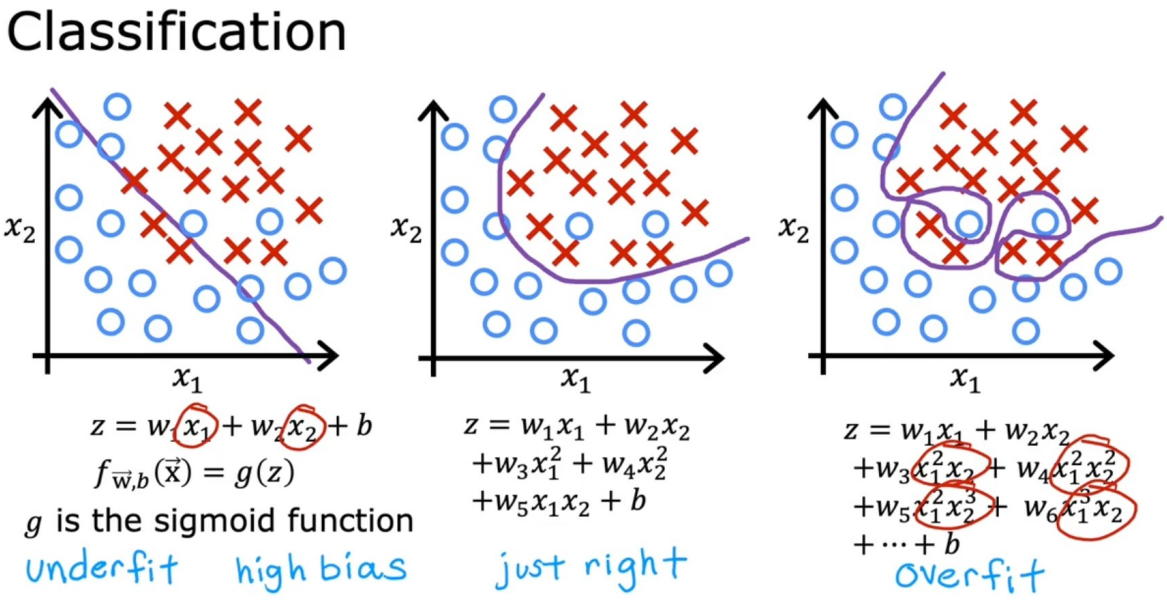
Definition 7.1.1 ▶ overfitting

- **overfitting**: the model is too complex (high variance)
- **underfitting**: the model is too simple (high bias)
- the model is just right: the model generalizes well (regularization)

regression



classification



7.2 Addressing overfitting

options

1. **increase the number of training examples**
2. **reduce the number of features**
3. **regularization**, reduce the size of the parameters.

understanding of regularization

By deacreasing the size of the parameters, we can reduce the complexity of the model. So the decision boundary will become more smooth, which will addressing overfitting.

We want to deacrease the size of certain parameters, but we don't know which ones. So we add a term to the cost function to penalize the size of the parameters.

> notice that the bias term b is not included in the regularization term.

7.3 Regularization

7.3.1 regularization for linear regression

cost function

Theorem 7.3.1 ▶ regularization cost function

we can modify the cost function as follows:

$$J(\mathbf{w}, b) = \frac{1}{2m} \sum_{i=1}^{m} \left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right) - y^{(i)} \right)^2 + \frac{\lambda}{2m} \sum_{j=1}^{n} w_j^2 \tag{7.1}$$

by adding the term $\frac{\lambda}{2m} \sum_{j=1}^{n} w_j^2$ to the cost function, we can penalize the size of the parameters.

gradient descent

Theorem 7.3.2 ▶ regularization gradient descent

$$w_j := w_j(1 - \alpha\frac{\lambda}{m}) - \alpha\frac{1}{m} \sum_{i=1}^{m} \left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right) - y^{(i)} \right) x_j^{(i)} \tag{7.2}$$

$$b := b - \alpha\frac{1}{m} \sum_{i=1}^{m} \left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right) - y^{(i)} \right) \tag{7.3}$$

understanding: the term $w_j(1 - \alpha\frac{\lambda}{m})$ will decrease the size of the parameter w_j each iteration. because we don't want to decrease the bias term b, the update of b remains the same.

7.3.2 regularization for logistic classification

cost function

Theorem 7.3.3 ▶ regularization cost function

we can modify the cost function as follows:

$$J(\mathbf{w},b) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)) + (1-y^{(i)})\log(1-f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2 \quad (7.4)$$

gradient descent

Theorem 7.3.4 ▶ regularization gradient descent

$$w_j := w_j(1-\alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)x_j^{(i)} \quad (7.5)$$

$$b := b - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right) \quad (7.6)$$

understanding: The formula is the same as the one for linear regression. But the cost function $f_{\mathbf{w},b}(x)$ is different.

Part II

Advanced Machine Learning Algorithms

8 Neural Networks Introduction

8.1 Neural networks intuition

8.1.1 Neuron and the brain

neurons is the basic unit of the brain, it is a cell that receives electrical and chemical signals from other neurons and processes them. The neuron has a cell body, dendrites, and an axon. The cell body contains the nucleus and other organelles. The dendrites are the input part of the neuron, they receive signals from other neurons. The axon is the output part of the neuron, it sends signals to other neurons. The axon is connected to the dendrites of other neurons through synapses. The synapse is the connection between the axon of one neuron and the dendrite of another neuron. The synapse is where the electrical and chemical signals are transmitted from one neuron to another. The brain is made up of billions of neurons that are connected to each other through synapses. The neurons communicate with each other by sending electrical and chemical signals through the synapses. This communication between neurons is what allows the brain to process information and perform complex tasks.

Simplified model of a neuron

A neuron can be modeled as a simple mathematical function that takes an **input** and produces an **output**.

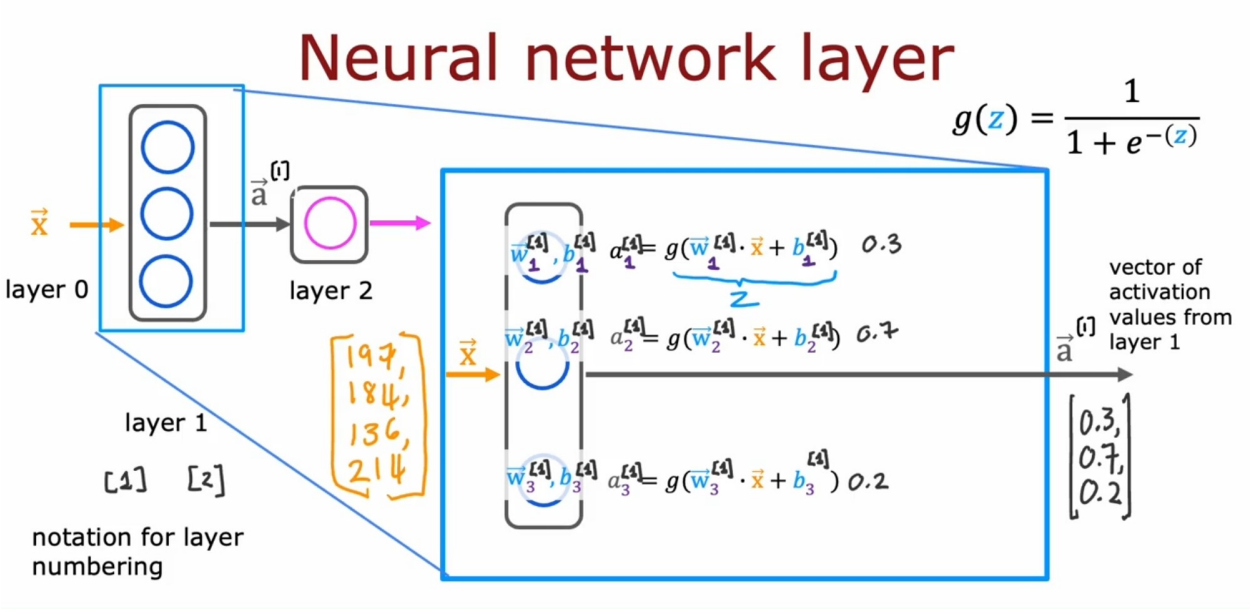
8.1.2 Demand prediction

Terminology

- **layer**: a collection of neurons that are connected to each other.
- **input layer**: the first layer of the neural network, it receives the input data.
- **output layer**: the last layer of the neural network, it produces the output data.
- **hidden layer**: a layer that is not the input or output layer.

- **neuron**: a node in the neural network that takes an input and produces an output.
- **activation function**: a function that determines the output of a neuron given its input.
- **activation**: the output of a neuron after applying the activation function. Activstions are higher level features.

In the neural network architecture, the input layer receives the input data, the hidden layers process the data, and the output layer produces the output data. And the neural networks can have multiple hidden layers, each layer can have multiple neurons, and each neuron can have multiple inputs and outputs. It is also called “multilayer perceptron”.



8.2 Neural networks model

Notation Every layer has a weight matrix $\mathbf{W}$ and a bias vector $\mathbf{b}$.

In the layer j, the weight matrix $\mathbf{W}^{[j]}$ has the dimension $n^{[j-1]} \times n^{[j]}$ and the bias vector $\mathbf{b}^{[j]}$ has the dimension $1 \times n^{[j]}$. Weights of each neuron $\mathbf{w}_i^{[j]}$ are column vectors not scalars. And the input and output vectors $\mathbf{a}$ are both row vectors.

$$\mathbf{W}^{[j]} = \begin{bmatrix} \mathbf{w}_1^{[j]} & \mathbf{w}_2^{[j]} & \cdots & \mathbf{w}_{n^{[j]}}^{[j]} \end{bmatrix} \tag{8.1}$$

$$\mathbf{b}^{[j]} = \begin{bmatrix} b_1^{[j]} & b_2^{[j]} & \cdots & b_n^{[j]} \end{bmatrix} \tag{8.2}$$

And the layer j takes in the input $\mathbf{a}^{[j-1]}$ and produces the output $\mathbf{a}^{[j]}$ (whose dimension is $n^{[j]}$, the number of neurons of layer j).

$$a_i^{[j]} = g(\mathbf{a}^{[j-1]} \cdot \mathbf{w}_i^{[j]} + b_i^{[j]}) \tag{8.3}$$

$$(\mathbf{a}^{[j]})^T = \begin{bmatrix} a_1^{[j]} \\ a_2^{[j]} \\ \vdots \\ a_n^{[j]} \end{bmatrix} = \begin{bmatrix} g(\mathbf{w}_1^{[j]}\mathbf{a}^{[j-1]} + b_1^{[j]}) \\ g(\mathbf{w}_2^{[j]}\mathbf{a}^{[j-1]} + b_2^{[j]}) \\ \vdots \\ g(\mathbf{w}_n^{[j]}\mathbf{a}^{[j-1]} + b_n^{[j]}) \end{bmatrix} \tag{8.4}$$

The dimension of the weight matrix $\mathbf{w}_i^{[j]}$ equals to the previous layer's number of neurons $n^{[j-1]}$, which is the dimension of $\mathbf{a}^{[j-1]}$. g is the activation function such as sigmoid function $g(x) = \frac{1}{1+e^{-x}}$.

Taking advantage of the vectorization method, the previous formula can be represented as

$$\mathbf{a}^{[j]} = g(\mathbf{a}^{[j-1]} \cdot \mathbf{W}^{[j]} + \mathbf{b}^{[j]}) \tag{8.5}$$

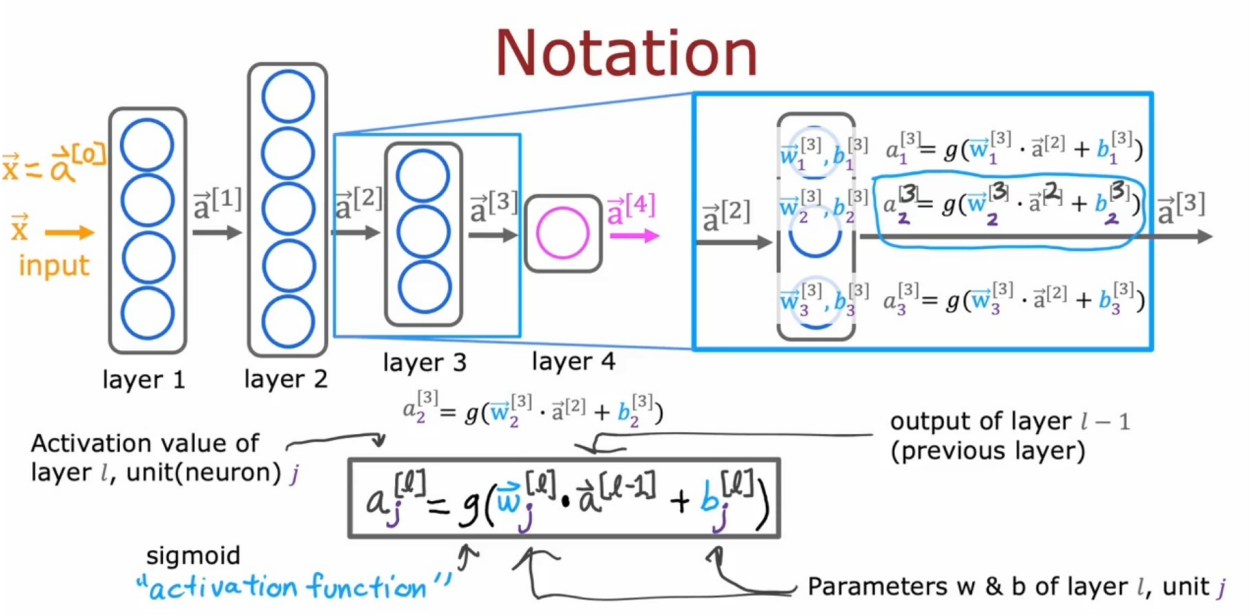
here $\cdot$ is the matrix multiplication, and $g(\mathbf{x})$ means apply g to each element in $\mathbf{x}$.

The numbers of neurons in each layer often decrease as we move from the input layer to the output layer.

And the neural network can do inference and predict category. By looking at the output layer, we can see the probability of each category. It's a scalar value, if the value is greater than 0.5, then the neural network predicts the certain category. And this procedure is called

"forward propagation".

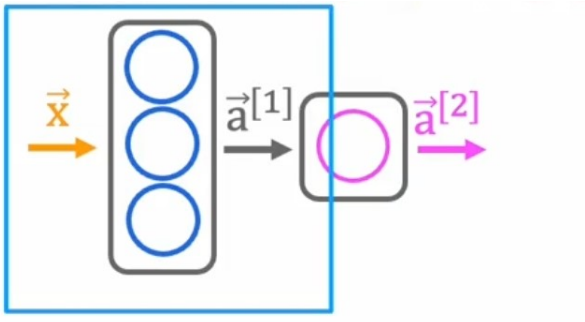
There are many things neural networks can do, such as image recognition, speech recognition, natural language processing, and many other tasks.



8.3 Implementation in TensorFlow

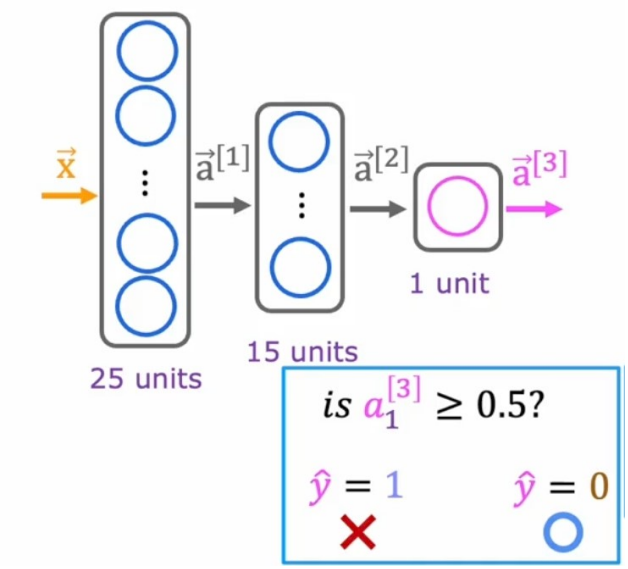
Inference

Assume you already have a trained model, and you want to use it to make predictions on new data.



```
x = np.array([[200.0, 17.0]])
layer_1 = Dense(units=3, activation='sigmoid')
a1 = layer_1(x)
```

```
    layer_2 = Dense(units=1, activation='sigmoid')
    a2 = layer_2(a1)

    if a2 >= 0.5:
        yhat = 1
    else:
        yhat = 0
```




```
    x = np.array([[0.0, 245.0, ..., 240.0, 0.0]])
    layer_1 = Dense(units=25, activation='sigmoid')
    a1 = layer_1(x)

    layer_2 = Dense(units=15, activation='sigmoid')
    a2 = layer_2(a1)

    layer_3 = Dense(units=1, activation='sigmoid')
    a3 = layer_3(a2)

    if a3 >= 0.5:
        yhat = 1
    else:
        yhat = 0
```

Data in TensorFlow

The data in TensorFlow is represented as tensors, which are multi-dimensional arrays. There is a little difference between the data in TensorFlow and the data in NumPy. But it is easy to convert the data between TensorFlow and NumPy.

```
x = np.array([[200.0, 17.0]])
layer_1 = Dense(units=3, activation='sigmoid')
a1 = layer_1(x)
# a1 : tf.Tensor([[0.2, 0.7, 0.3]], shape=(1, 3), dtype=float32)
# a1.numpy() : array([[0.2, 0.7, 0.3]], dtype=float32)
layer_2 = Dense(units=1, activation='sigmoid')
a2 = layer_2(a1)
# a2 : tf.Tensor([[0.8]], shape=(1, 1), dtype=float32)
# a2.numpy() : array([[0.8]], dtype=float32)
```

The example of "Coffee Roasting" in optional lab, you may find helpful.

8.4 Building a neural network in TensorFlow

How to build a neural network in TensorFlow?

```
model = Sequential([
    Dense(units=3, activation='sigmoid'),
    Dense(units=1, activation='sigmoid')
])

x = np.array([[200.0, 17.0],
              [100.0, 24.0],
              [150.0, 20.0],
              [110.0, 22.0]])
y = np.array([1, 0, 1, 0])

model.compile(...)
model.fit(x, y)

model.predict(x_new)
```

8.5 Forward propagation

How to implement forward propagation in python?

```
def dense(a_in, W, b):
    units = W.shape[1]
    a_out = np.zeros(units)
    for j in range(units):
        w = W[:, j]
        z = np.dot(w, a_in) + b[j]
        a_out[j] = sigmoid(z)
    return a_out
```

Dense layer vectorized implementation

```
def dense(a_in, W, b):
    z = np.dot(a_in, W) + b
    # z = np.matmul(a_in, W) + b
    a_out = sigmoid(z)
    return a_out
```

Sequential forward

```
def sequential_forward(x, model):
    a = x
    for layer in model:
        a = dense(a, layer['W'], layer['b'])
    return a
```

9 Neural Networks Training

9.1 Training

Train a neural network in TensorFlow.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
model = Sequential([
    Dense(units=25, activation='sigmoid'),
    Dense(units=15, activation='sigmoid'),
    Dense(units=1, activation='sigmoid')
])
from tensorflow.keras.loss import BinaryCrossentropy
model.compile(loss=BinaryCrossentropy())
model.fit(X, Y, epochs=100)
```

9.2 Details

Model Training Details

1. specify how to compute the output given input x and parameters w, b (define model)

$$f_{\mathbf{w},b}(\mathbf{x}) = ?$$

2. specify how to compute the loss given the output and the target

$$L(f_{\mathbf{w},b}(\mathbf{x}), y) = ?$$

$$J(\mathbf{w}, b) = \frac{1}{m} \sum_{i=1}^{m} L(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)})$$

3. how to update the parameters to minimize the loss

logistic regression	neural network
1. `z = np.dot(x, w) + b` `f_x = 1 / (1 + np.exp(-z))`	1. `model = Sequential([...])`
2. **Logistic Loss** `loss = -y * np.log(f_x) - (1 - y) * np.log(1 - f_x)`	2. **Binary Cross Entropy** `model.compile(loss=BinaryCrossentropy())`
3. `w = w - alpha * dw` `b = b - alpha * db`	3. `model.fit(X, Y, epochs=100)`

Create the model

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(units=25, activation='sigmoid'),
    Dense(units=15, activation='sigmoid'),
    Dense(units=1, activation='sigmoid')
])
```

Loss and cost function

logistic loss also known as binary cross entropy:

$$L(f_{\mathbf{w},b}(\mathbf{x}), y) = -y \log(f_{\mathbf{w},b}(\mathbf{x})) - (1-y) \log(1 - f_{\mathbf{w},b}(\mathbf{x}))$$

```
from tensorflow.keras.loss import BinaryCrossentropy
model.compile(loss=BinaryCrossentropy())
```

If you are predicting numbers and not classes, you can use mean squared error:

```
from tensorflow.keras.loss import MeanSquaredError
model.compile(loss=MeanSquaredError())
```

Gradient descent

The tensorflow model will automatically compute the gradients and update the weights and biases using **backpropagation**

```
model.fit(X, Y, epochs=100)
```

9.3 Activation functions

Definition 9.3.1 ▶ Linear Activation

Linear activation is the simplest activation function. It is also called "no activation function".

$$g(z) = z \tag{9.1}$$

Definition 9.3.2 ▶ Sigmoid Activation

Sigmoid activation squashes the output to be between 0 and 1.

$$g(z) = \frac{1}{1 + e^{-z}} \tag{9.2}$$

Definition 9.3.3 ▶ ReLU Activation

Relu activation (Rectified Linear Unit) is the most popular activation function.

$$g(z) = \max(0, z) \tag{9.3}$$

Choosing activation functions

Output layer

If the output should be between 0 and 1, use sigmoid activation. If the output can be negative, use linear activation. If the output is non-negative, use ReLU activation. And if the

output is a multi-class classification, use softmax activation.

- For binary classification, use sigmoid activation.
- For multi-class classification, use softmax activation.
- For regression, use linear activation.

Hidden layers

ReLU activation is the most popular activation function for hidden layers.

Sigmoid function has two "flat" zones, which can slow down learning. And compared to ReLU, sigmoid are more computationally expensive.

So, usually we use ReLU for hidden layers.

> If use all the layers with Linear activation, the whole network will be equivalent to a single layer with linear activation, which is same as linear regression.
>
> If use all the hidden layers with Linear activation, and the output layer with sigmoid activation, the whole network will be equivalent to a single layer with sigmoid activation, which is same as logistic regression.

9.4 Multiclass Classification

Softmax regression

Logistic regression (2 possible outcomes)

$$a_1 = P(y = 1|x) = \frac{1}{1 + e^{-(w^T x + b)}}$$
$$a_2 = P(y = 0|x) = 1 - a_1$$

Softmax regression (N possible outcomes)

$$z_j = w_j^T x + b_j$$
$$a_j = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}}$$

note: $a_1 + a_2 + \cdots + a_N = 1$

Cost

Logistic regression

$$L(a, y) = \begin{cases} -\log(a_1) & \text{if } y = 1 \\ -\log(a_2) & \text{if } y = 0 \end{cases} = -y \log(a_1) - (1 - y) \log(1 - a_1)$$

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(a^{(i)}, y^{(i)})$$

Softmax regression

$$\begin{aligned} a_1 = P(y = 1|x) &= \frac{e^{z_1}}{e^{z_1} + e^{z_2} + \cdots + e^{z_N}} \\ a_2 = P(y = 2|x) &= \frac{e^{z_2}}{e^{z_1} + e^{z_2} + \cdots + e^{z_N}} \\ &\vdots \\ a_N = P(y = N|x) &= \frac{e^{z_N}}{e^{z_1} + e^{z_2} + \cdots + e^{z_N}} \end{aligned}$$

Cross entropy loss

$$\text{loss} = \begin{cases} -\log(a_1) & \text{if } y = 1 \\ -\log(a_2) & \text{if } y = 2 \\ & \vdots \\ -\log(a_N) & \text{if } y = N \end{cases} = -\log(a_j) \quad \text{if } y = j$$

```
from tensorflow.keras.loss import CategoricalCrossentropy
model.compile(loss=CategoricalCrossentropy())
```

Template

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.loss import SparseCategoricalCrossentropy

model = Sequential([
    Dense(units=25, activation='relu'),
    Dense(units=15, activation='relu'),
    Dense(units=10, activation='softmax')
])
model.compile(loss=SparseCategoricalCrossentropy())
model.fit(X, Y, epochs=100)
```

This verison is not recommended, because it is not efficient.

Here is a improved version:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.loss import SparseCategoricalCrossentropy

model = Sequential([
    Dense(units=25, activation='relu'),
    Dense(units=15, activation='relu'),
    Dense(units=10, activation='linear')
])
model.compile(..., loss=SparseCategoricalCrossentropy(from_logits=True))
model.fit(X, Y, epochs=100)
```

Predication:

```
logits = model.predict(new_data)
probabilities = tf.nn.softmax(logits)
```

Explaination

Numerical Roundoff Error

Because the computation of softmax involves exponentiation, it can be numerically unstable. There exists a way of TensorFlow to compute the softmax and the cross-entropy loss in a single step, which is more numerically stable.

- Change the output layer activation to **linear**.
- Add `from_logits=True` to the loss function.
- Because the result haven't been processed with Softmax, Use `tf.nn.softmax` to compute the probabilities.

`from_logits=True` means that the output of the model is not probabilities, but logits. By applying `from_logits=True`, the loss function will automatically apply softmax to the output of the model before computing the loss. The reason for chaning the output layer activation to linear is and applying from_logits=True is that TensorFlow can optimize the computation of the softmax and the cross-entropy loss, which is more numerically stable.

The same can be done for logistic regression, by changing the output layer activation to linear and using `BinaryCrossentropy(from_logits=True)`.

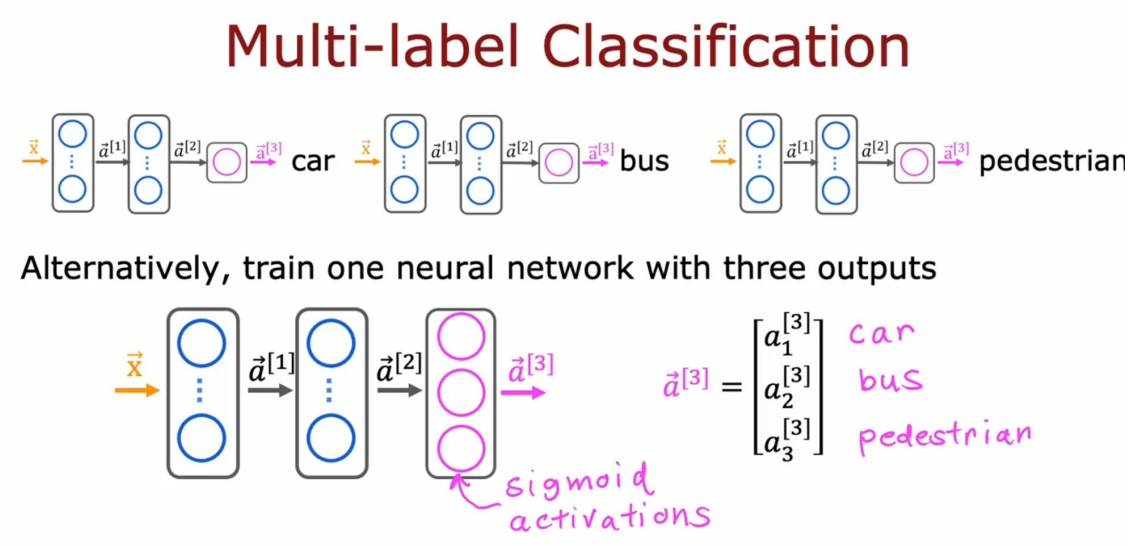
```
model = Sequential([
    Dense(units=25, activation='relu'),
    Dense(units=15, activation='relu'),
    Dense(units=1, activation='linear')
])
model.compile(..., loss=BinaryCrossentropy(from_logits=True))
model.fit(X, Y, epochs=100)

logits = model.predict(new_data)
probabilities = tf.nn.sigmoid(logits)
```

9.5 Multi-label Classification

Multilable classification is a generalization of multiclass classification, where each instance can be assigned multiple labels.

differences between multiclass and multilabel classification

- In multiclass classification, each instance is assigned to one and only one class.
- In multilabel classification, each instance can be assigned to multiple classes.

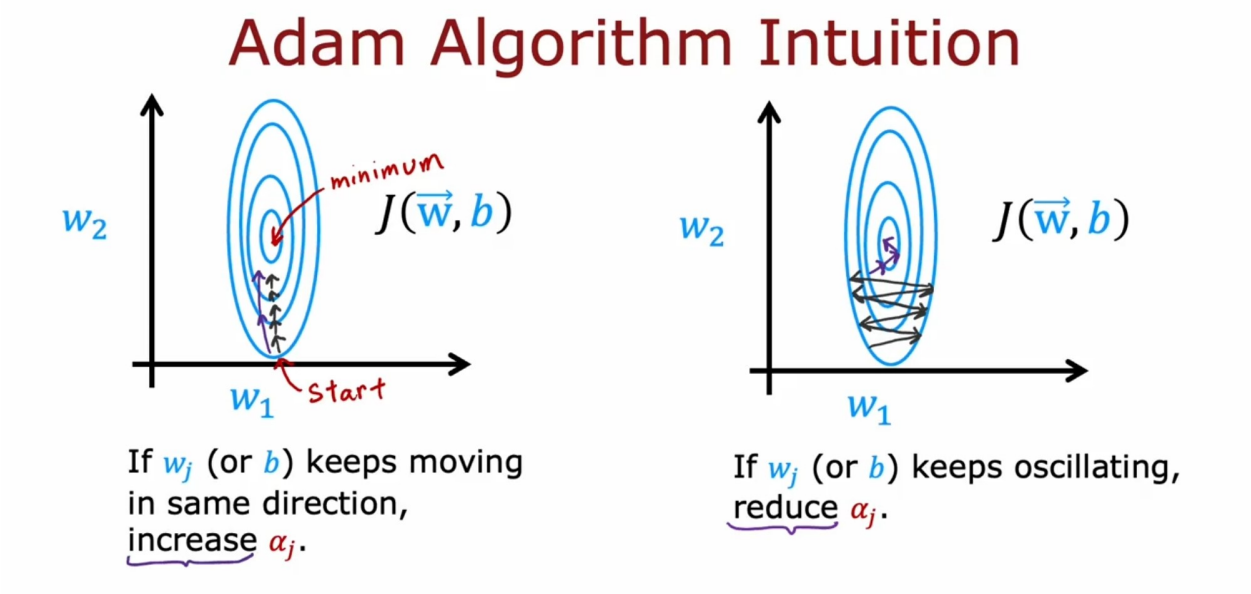


In this example, each image can be assigned to multiple labels. For example, an image can be assigned to both “include car”, “include bus”, and “include pedestrian”. We can build 3 separate binary classifiers, one for each label. Also, we can build a single neural network with 3 output units, one for each label. (multi-label classification) Usually, we use the sigmoid activation function for the output layer in multi-label classification while we use the softmax activation function in multi-class classification.

> The output of multiclass classification is a probability distribution over the classes, so its sum must be 1. But the output of multilabel classification is not a probability distribution, so its sum is not 1.

9.6 Adam Optimizer

Adam is an optimization algorithm that can be used instead of the classical gradient descent procedure to update network weights iteratively based on training data.



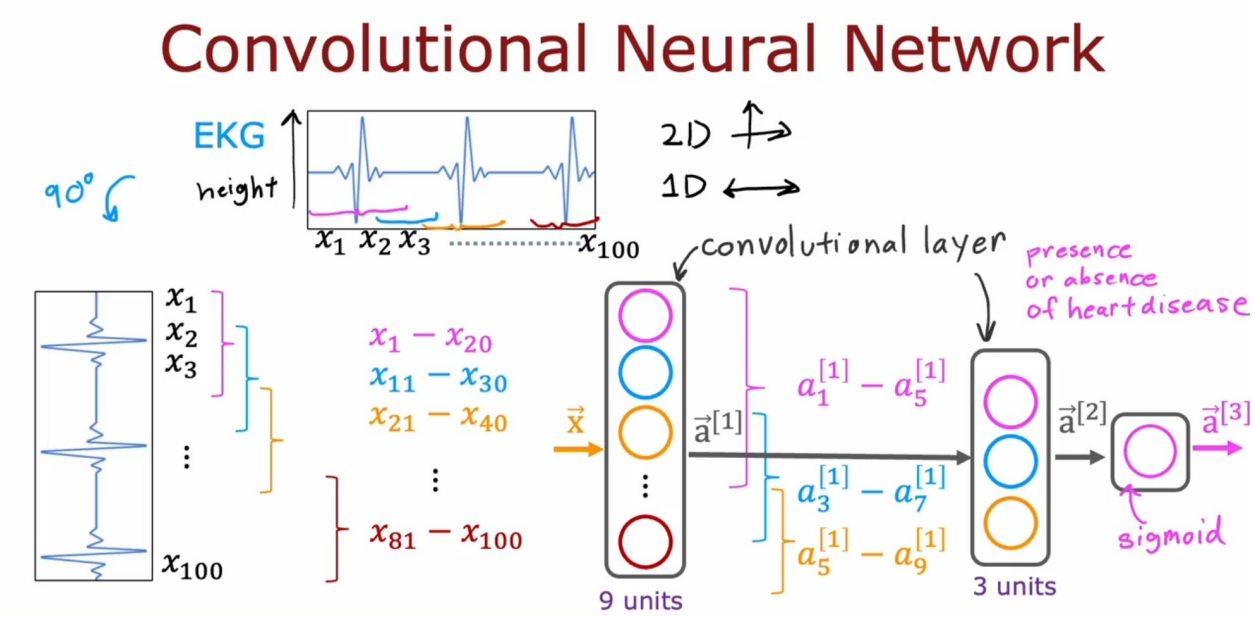
By changing the learning rate, we can achieve better performance.

```
model = Sequential([
    Dense(units=25, activation='relu'),
    Dense(units=15, activation='relu'),
    Dense(units=1, activation='linear')
])
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
 loss=BinaryCrossentropy(from_logits=True))
model.fit(X, Y, epochs=100)
```

9.7 Additional layer types

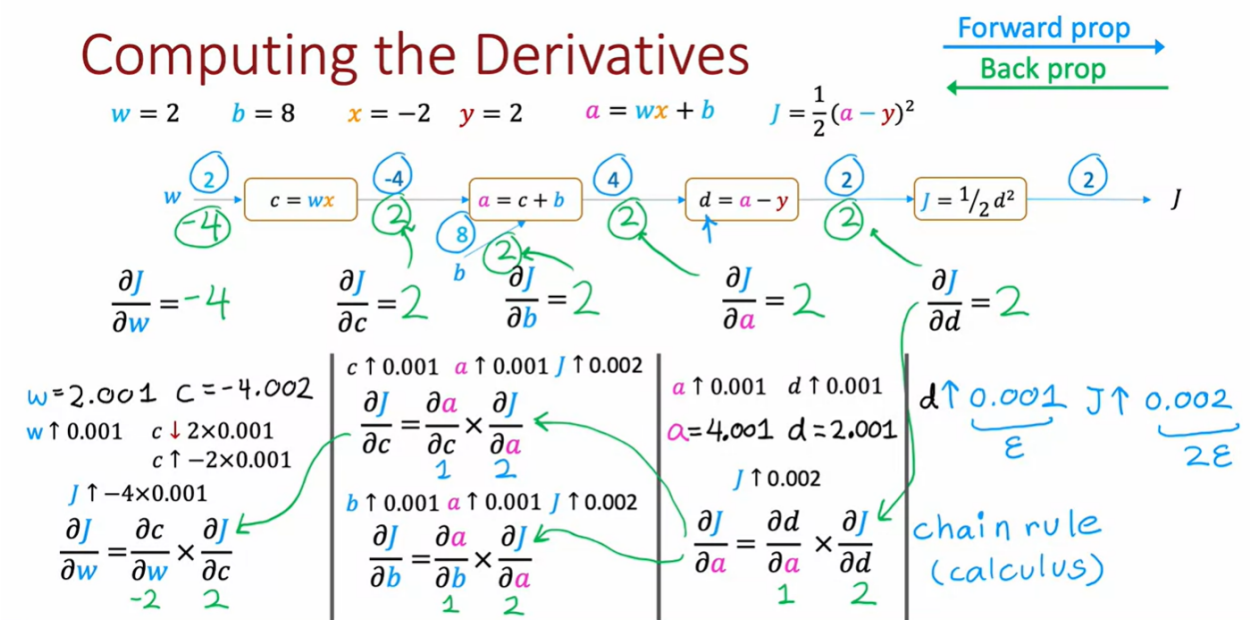
Dense layer: Each neuron output is a function of all the inputs.

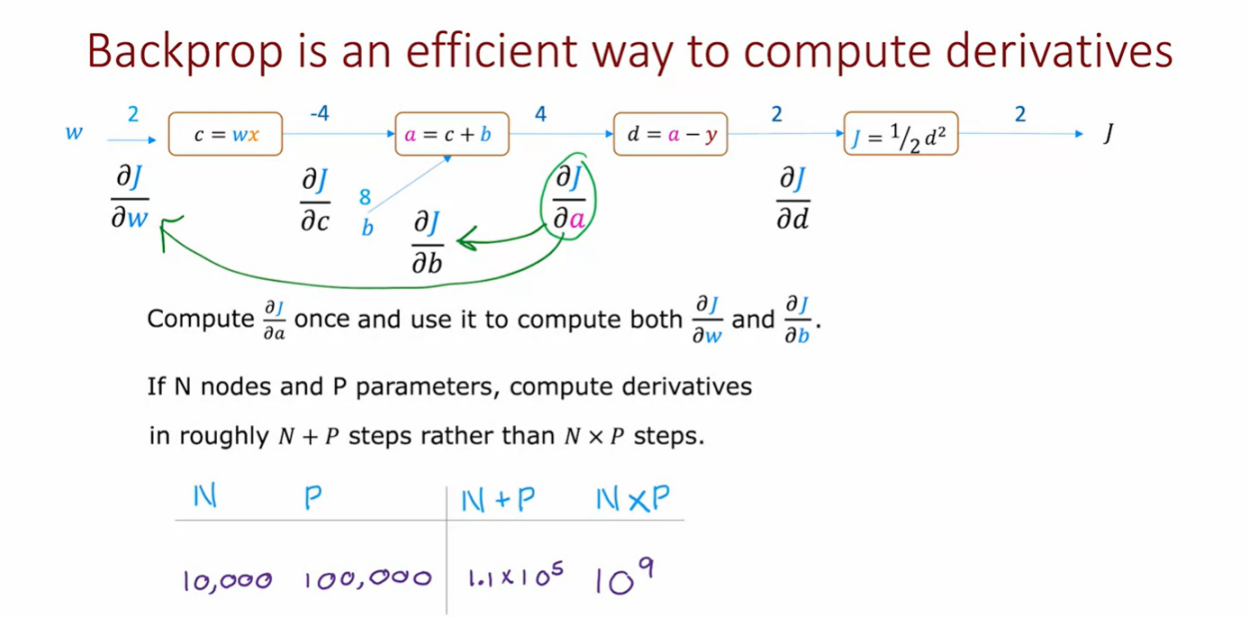
Convolutional layer: Each neuron output is a function of a subset of the inputs. Why? 1. Faster computation. 2. Need less training data (less prone to overfitting).



9.8 Computation graph

Computation graph is a way to represent the computation of a neural network. We use forward propagation to compute the output of the neural network and use backpropagation to compute the gradients of the loss function with respect to the parameters of the neural network. The essence of backpropagation is "**Chain Rule**" in calculus.





When computing derivatives for a two-layer neural network using backpropagation, the process proceeds as follows:

Forward Propagation:

- Input to hidden layer computation: $h = f(W_1 x + b_1)$, where W_1 is the weight matrix for the hidden layer and b_1 is the bias vector.
- Hidden to output layer computation: $y = g(W_2 h + b_2)$, where W_2 is the weight matrix for the output layer and b_2 is the bias vector.
- Loss computation: $L = L(y, y_{\text{true}})$, where y_{true} is the true target output.

Backward Propagation: The goal of backpropagation is to compute the derivatives of the loss function L with respect to all parameters, i.e., $\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial b_1}, \frac{\partial L}{\partial W_2}, \frac{\partial L}{\partial b_2}$.

- **Compute derivatives at the output layer:**

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial W_2}$$

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial b_2}$$

- **Compute derivatives at the hidden layer:**

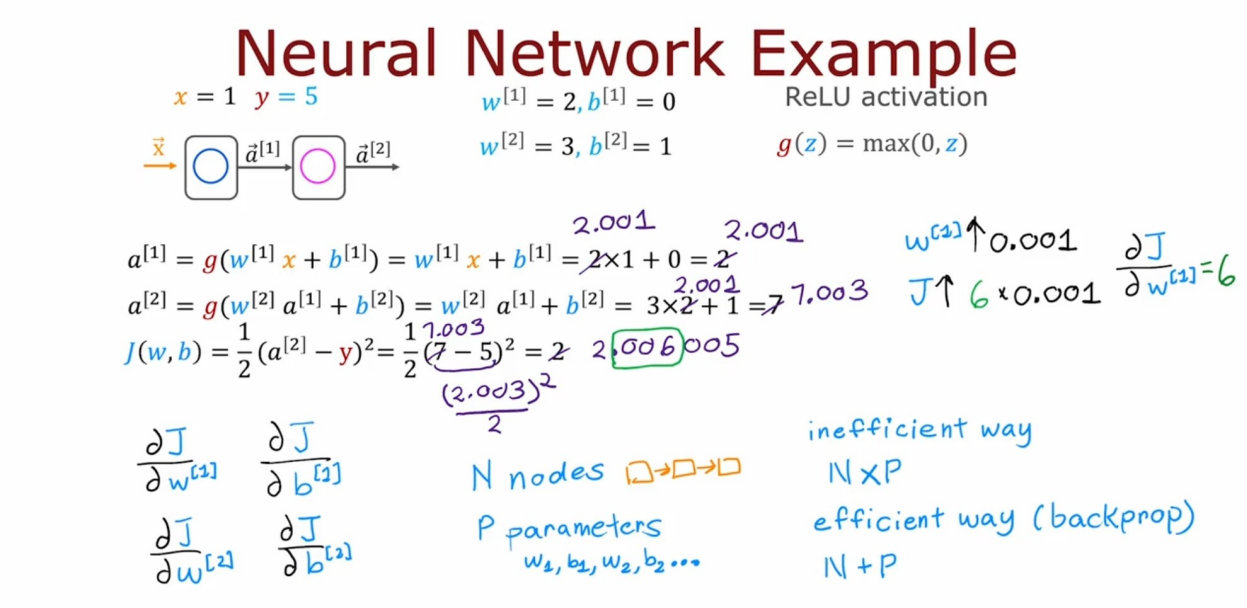
$$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial h}$$

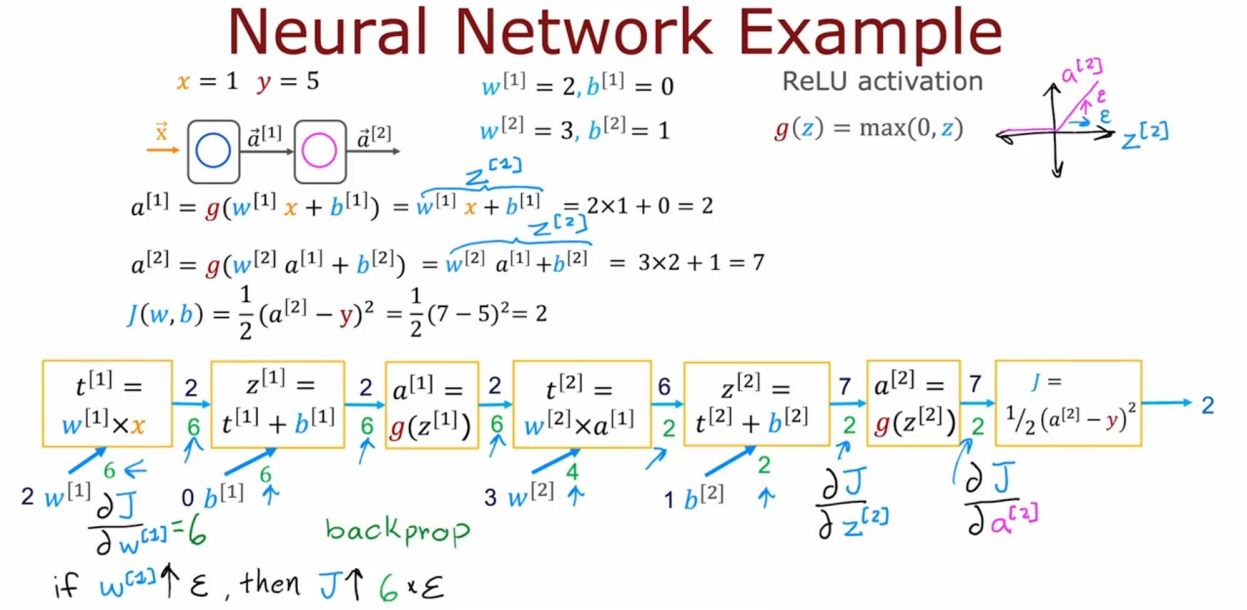
$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial W_1}$$

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial b_1}$$

Here, derivatives are computed layer by layer, starting from the output layer and propagating backwards to the hidden layer. The chain rule is applied to propagate errors back through the network.

Back propagation is an efficient way to compute the gradients of the loss function with respect to the parameters of the neural network. When there is N nodes and P parameters in the network, the back propagation algorithm only need to compute $O(N+P)$ operations instead of $O(NP)$ operations. The N steps are used to compute the gradients of the loss function with respect to the output of each node ($\frac{\partial J}{\partial a^{[i]}}$), and the P steps are used to compute the gradients of the loss function with respect to each parameter ($\frac{\partial J}{\partial w}$ or $\frac{\partial J}{\partial b}$).




Neural Network Example
$x = 1 \quad y = 5$
$w^{[1]} = 2, b^{[1]} = 0$
$w^{[2]} = 3, b^{[2]} = 1$
ReLU activation
$g(z) = \max(0, z)$
$a^{[1]} = g(w^{[1]} x + b^{[1]}) = w^{[1]} x + b^{[1]} = 2\times1 + 0 = 2$
$a^{[2]} = g(w^{[2]} a^{[1]} + b^{[2]}) = w^{[2]} a^{[1]} + b^{[2]} = 3\times2 + 1 = 7$
$J(w,b) = \frac{1}{2}(a^{[2]} - y)^2 = \frac{1}{2}(7-5)^2 = 2$
$t^{[1]} = w^{[1]}\times x$
$z^{[1]} = t^{[1]} + b^{[1]}$
$a^{[1]} = g(z^{[1]})$
$t^{[2]} = w^{[2]}\times a^{[1]}$
$z^{[2]} = t^{[2]} + b^{[2]}$
$a^{[2]} = g(z^{[2]})$
$J = \frac{1}{2}(a^{[2]} - y)^2$
backprop
if $w^{[1]} \uparrow \varepsilon$, then $J \uparrow 6\times\varepsilon$

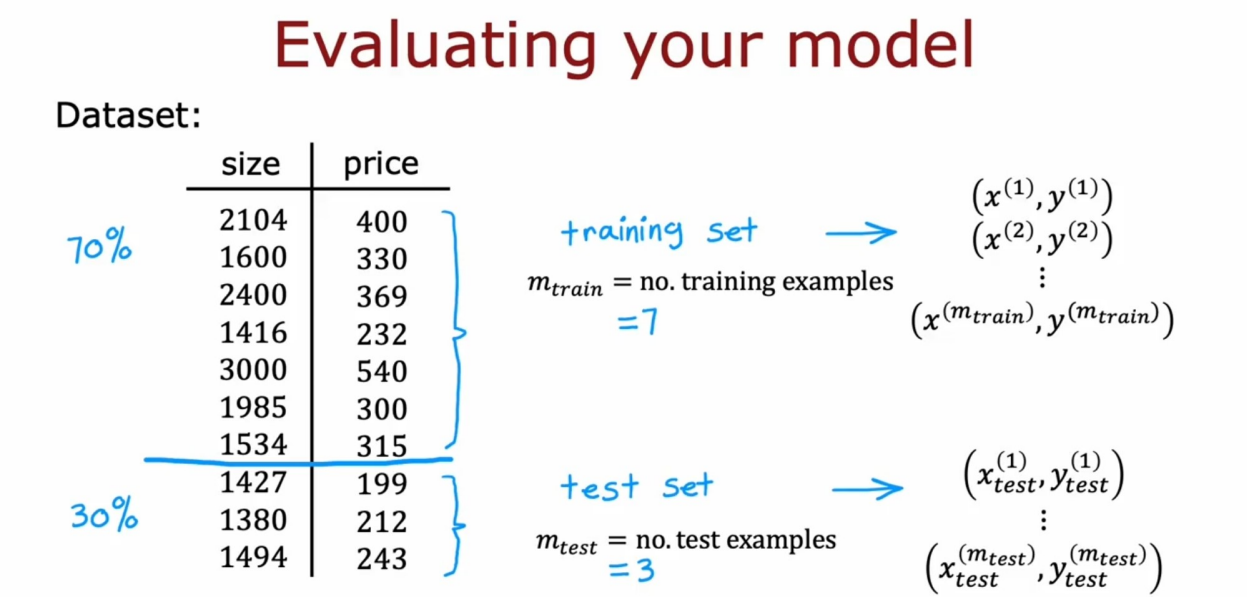
10

Advice for applying machine learning

10.1 Evaluating a model

Training and testing set

Split the data into two sets: a training set and a testing set. The training set is used to train the model, and the testing set is used to evaluate the model. The testing set should be large enough to give a good estimate of the model's performance. A common split is 80% training and 20% testing.



Compute the error:

For regression:

- $m_{train} = m_1, m_{test} = m_2$
- Fit parameters by minimizing the error cost function on the training set:

$$J(\mathbf{w}, b) = \frac{1}{2m_1} \sum_{i=1}^{m_1} (f_{\mathbf{w},b}(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m_1} \sum_{i=1}^{n} w_j^2$$

note: the second term is the regularization term.

- Compute the error on the test set:

$$J_{test}(\mathbf{w}, b) = \frac{1}{2m_2} \sum_{i=1}^{m_2} \left(f_{\mathbf{w},b}(x^{(i)}) - y^{(i)} \right)^2$$

note: the regularization term is not included in the test error.

- compute the training error:

$$J_{train}(\mathbf{w}, b) = \frac{1}{2m_1} \sum_{i=1}^{m_1} \left(f_{\mathbf{w},b}(x^{(i)}) - y^{(i)} \right)^2$$

For classification:

- $m_{train} = m_1, m_{test} = m_2$
- Fit parameters by minimizing the error cost function on the training set:

$$J(\mathbf{w}, b) = -\frac{1}{m_1} \sum_{i=1}^{m_1} \left[y^{(i)} \log \left(f_{\mathbf{w},b}(x^{(i)}) \right) + (1 - y^{(i)}) \log \left(1 - f_{\mathbf{w},b}(x^{(i)}) \right) \right] + \frac{\lambda}{2m_1} \sum_{i=1}^{n} w_j^2$$

note: the second term is the regularization term.

- Compute the error on the test set:

$$J_{test}(\mathbf{w}, b) = -\frac{1}{m_2} \sum_{i=1}^{m_2} \left[y^{(i)} \log \left(f_{\mathbf{w},b}(x^{(i)}) \right) + (1 - y^{(i)}) \log \left(1 - f_{\mathbf{w},b}(x^{(i)}) \right) \right]$$

- compute the training error:

$$J_{train}(\mathbf{w}, b) = -\frac{1}{m_1} \sum_{i=1}^{m_1} \left[y^{(i)} \log \left(f_{\mathbf{w},b}(x^{(i)}) \right) + (1 - y^{(i)}) \log \left(1 - f_{\mathbf{w},b}(x^{(i)}) \right) \right]$$

There is another way to compute the error for classification, which is more common in practice. The testing/training error is the fraction of the test/train set that has been misclassified:

- Compute the error on the test set:

$$J_{test}(\mathbf{w}, b) = \frac{1}{m_2} \sum_{i=1}^{m_2} \mathbf{1}\{y^{(i)} \neq \text{prediction}\}$$

- compute the training error:

$$J_{train}(\mathbf{w}, b) = \frac{1}{m_1} \sum_{i=1}^{m_1} \mathbf{1}\{y^{(i)} \neq \text{prediction}\}$$

where $\mathbf{1}\{y^{(i)} \neq \text{prediction}\}$ is an indicator function that is 1 if the prediction is wrong and 0 if the prediction is correct.
This is called the 0/1 misclassification error.

Usually, the training error is lower than the test error, because the model is trained to minimize the training error.

model selection

1. $f_{\mathbf{w},b} = w_1 x + b$	1. $w^{\langle 1 \rangle}, b^{\langle 1 \rangle}$	1. $J_{\text{test}}(w^{\langle 1 \rangle}, b^{\langle 1 \rangle})$
2. $f_{\mathbf{w},b} = w_1 x + w_2 x^2 + b$	2. $w^{\langle 2 \rangle}, b^{\langle 2 \rangle}$	2. $J_{\text{test}}(w^{\langle 2 \rangle}, b^{\langle 2 \rangle})$
3. $f_{\mathbf{w},b} = w_1 x + w_2 x^2 + w_3 x^3 + b$	3. $w^{\langle 3 \rangle}, b^{\langle 3 \rangle}$	3. $J_{\text{test}}(w^{\langle 3 \rangle}, b^{\langle 3 \rangle})$
$\vdots$	$\vdots$	$\vdots$
10. $f_{\mathbf{w},b} = w_1 x + w_2 x^2 + \cdots + w_{10} x^{10} + b$	10. $w^{\langle 10 \rangle}, b^{\langle 10 \rangle}$	10. $J_{\text{test}}(w^{\langle 10 \rangle}, b^{\langle 10 \rangle})$

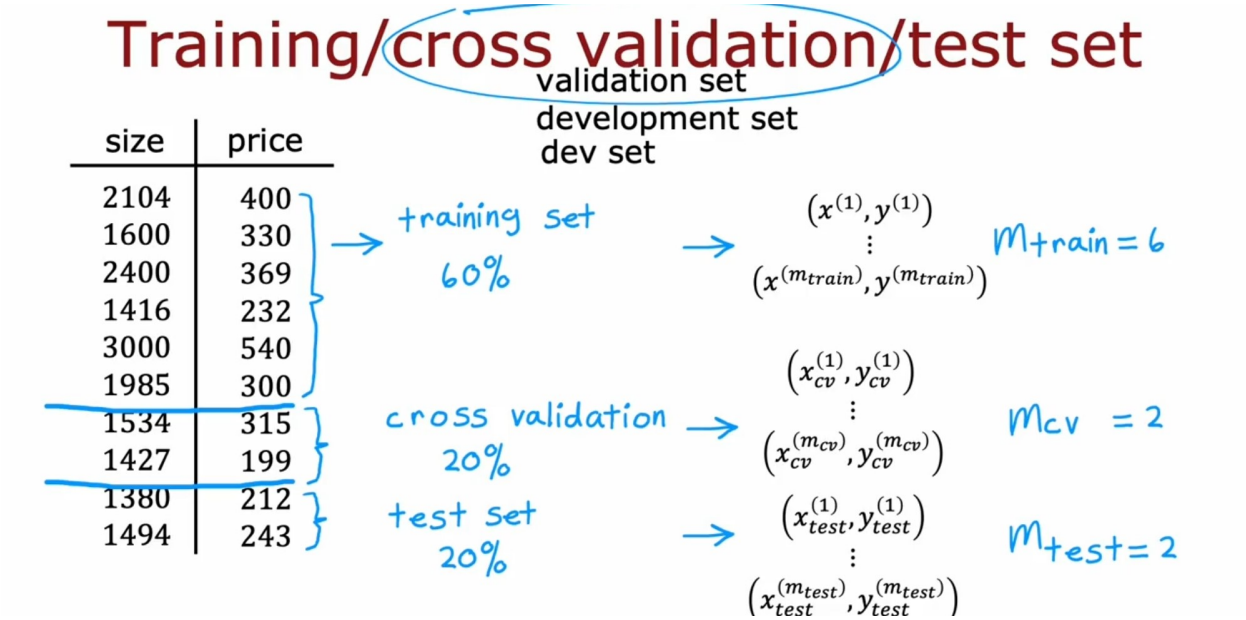
Assume that the $J_{\text{test}}(w^{\langle 5 \rangle}, b^{\langle 5 \rangle})$ is the smallest, then we choose the model with $d = 5$.

However, this method has flaws. If we want to get the generalization error, the problem is that $J_{\text{test}}(w^{\langle 5 \rangle}, b^{\langle 5 \rangle})$ is likely to be an overly optimistic estimate of generalization error(how well the model will perform on new data), which is to say $J_{\text{test}}(w^{\langle 5 \rangle}, b^{\langle 5 \rangle}) <$ generalization error.

This is because the extra parameter d was chosen using the test set. Just like choosing w, b from the training set, w, b are overly optimistic estimate of the generalization error.

Cross-validation

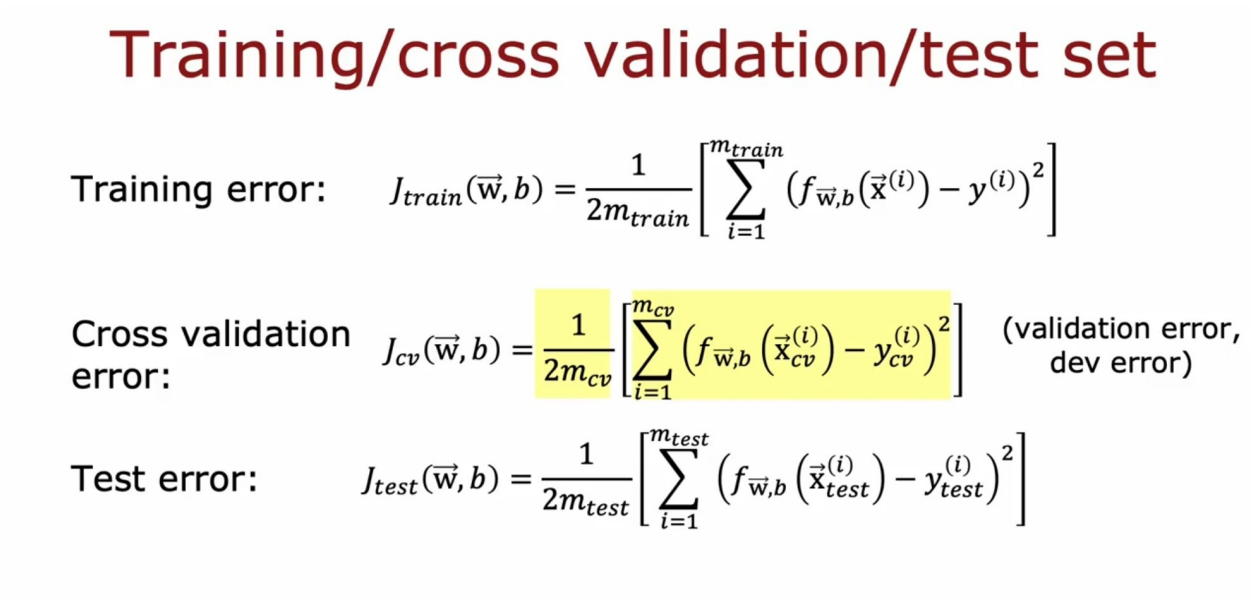
Split the data into three sets: a training set, a cross-validation set, and a testing set.



Compute the error:

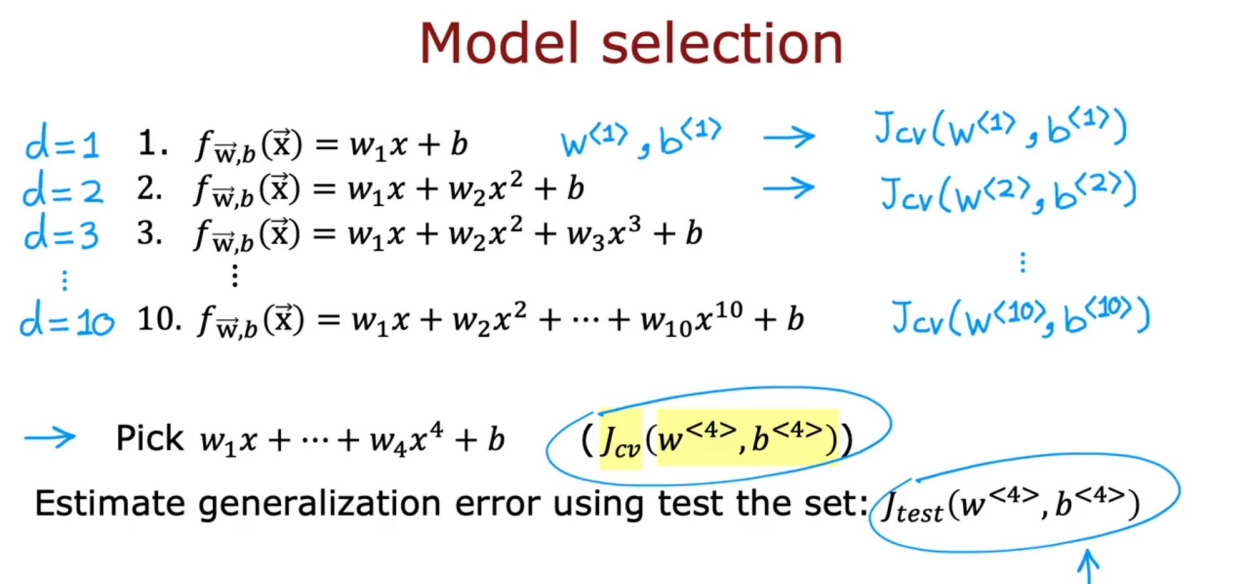
The same method as before, but now we will compute the cross-validation error:

$$J_cv(\mathbf{w}, b) = \frac{1}{2m_{cv}} \sum_{i=1}^{m_{cv}} \left(f_{\mathbf{w},b}(x_{cv}^{(i)}) - y_{cv}^{(i)} \right)^2$$

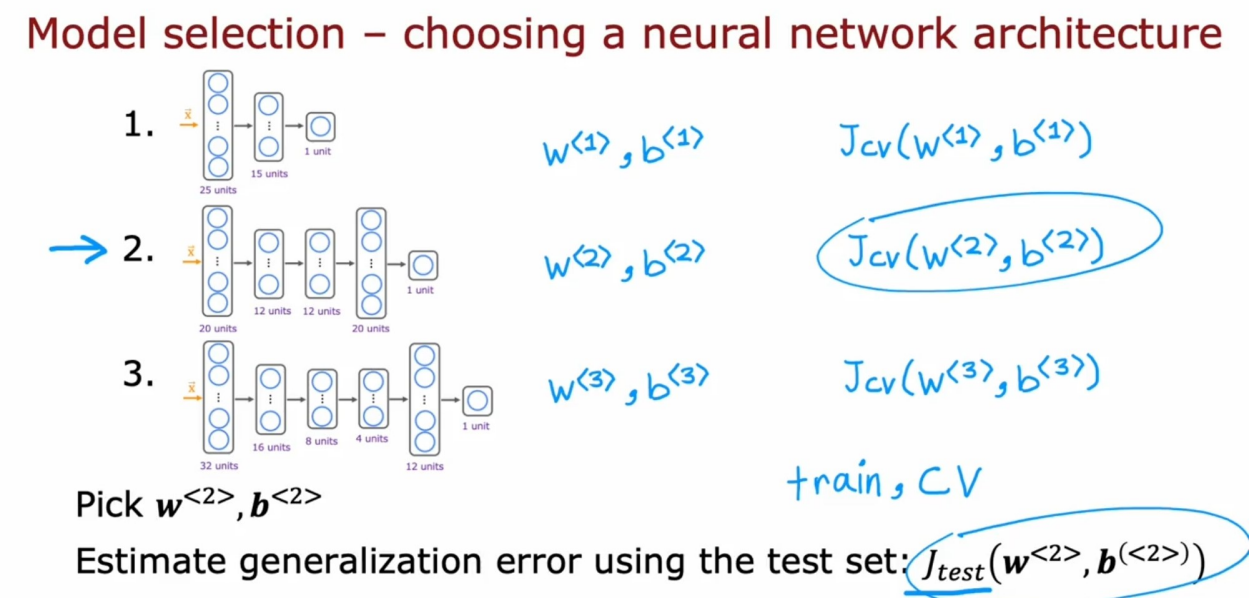


Model selection:

Most parts are the same as before, but now we will choose the model with the smallest cross-validation error.



It also can be applied to neural networks.



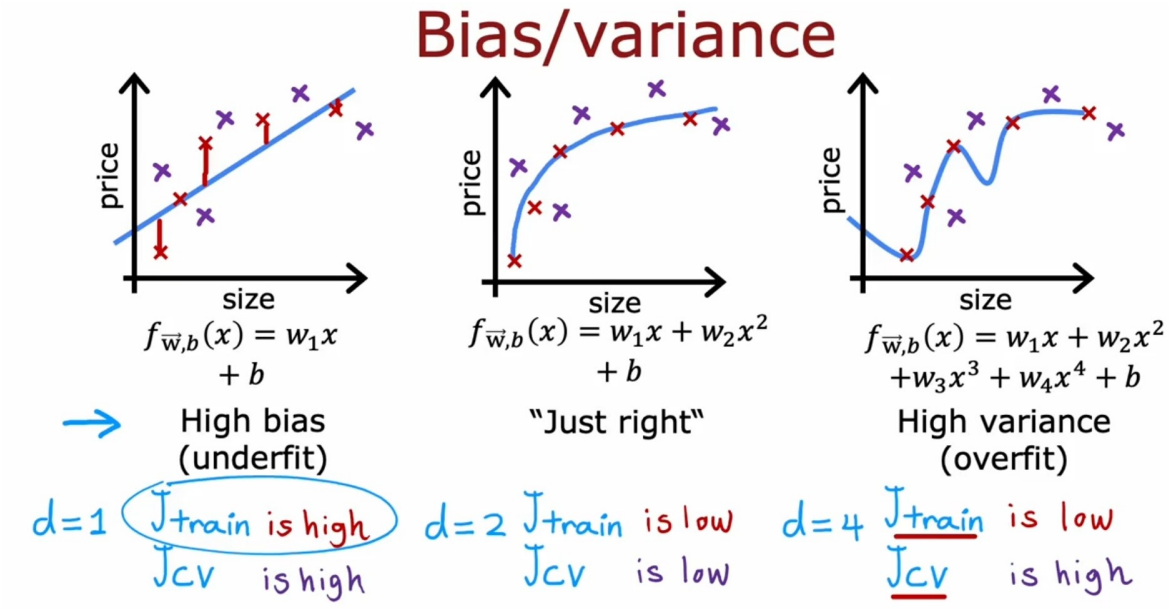
10.2 Bias and variance

Definition 10.2.1 ▸ Bias

Bias refers to the error that is introduced by approximating a real-life problem by a much simpler model, which may be extremely complex. Often, the model is too simple to capture the underlying structure of the data. This situation is called **underfitting**. And the indicator of bias is the training error. If the training error is high, then the model may have high bias.

Definition 10.2.2 ▸ Variance

Variance refers to the error that is introduced by approximating a real-life problem by a much more complex model. Often, the model is too complex to capture the underlying structure of the data. This situation is called **overfitting**. And the indicator of variance is the gap between the training error and the cross-validation error. If the gap is large, then the model may have high variance.



- High Bias (underfit):

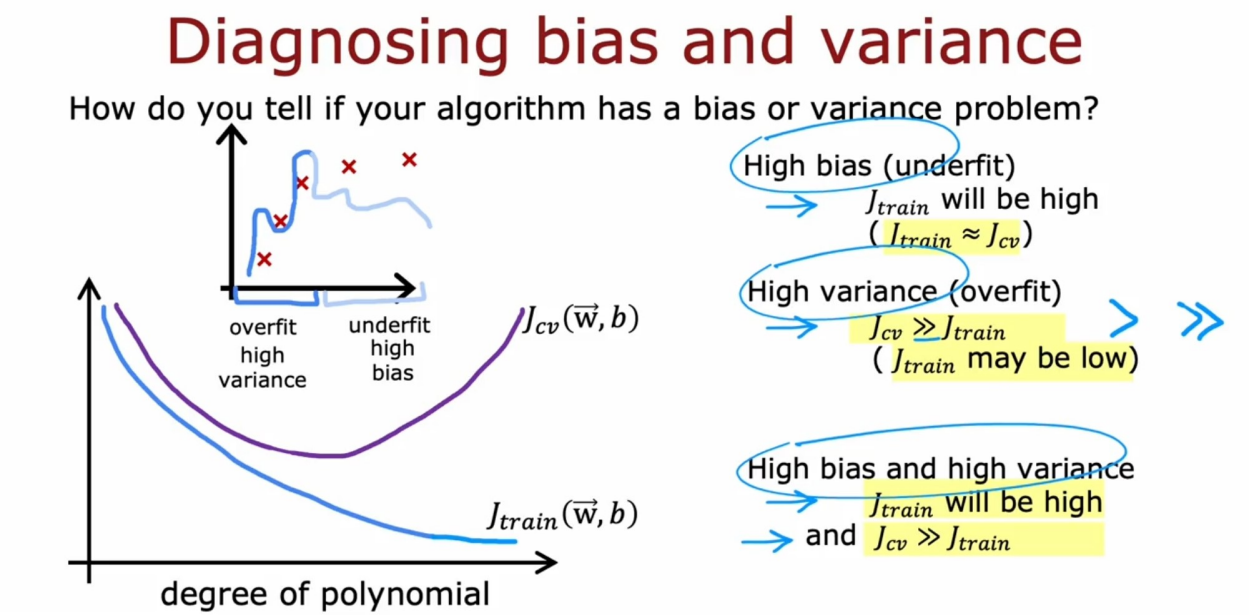
  J_{train} will be high and $J_{\text{cv}} \approx J_{\text{train}}$.

- High Variance (overfit):

  $J_{\text{cv}} \gg J_{\text{train}}$ and J_{train} may be low.

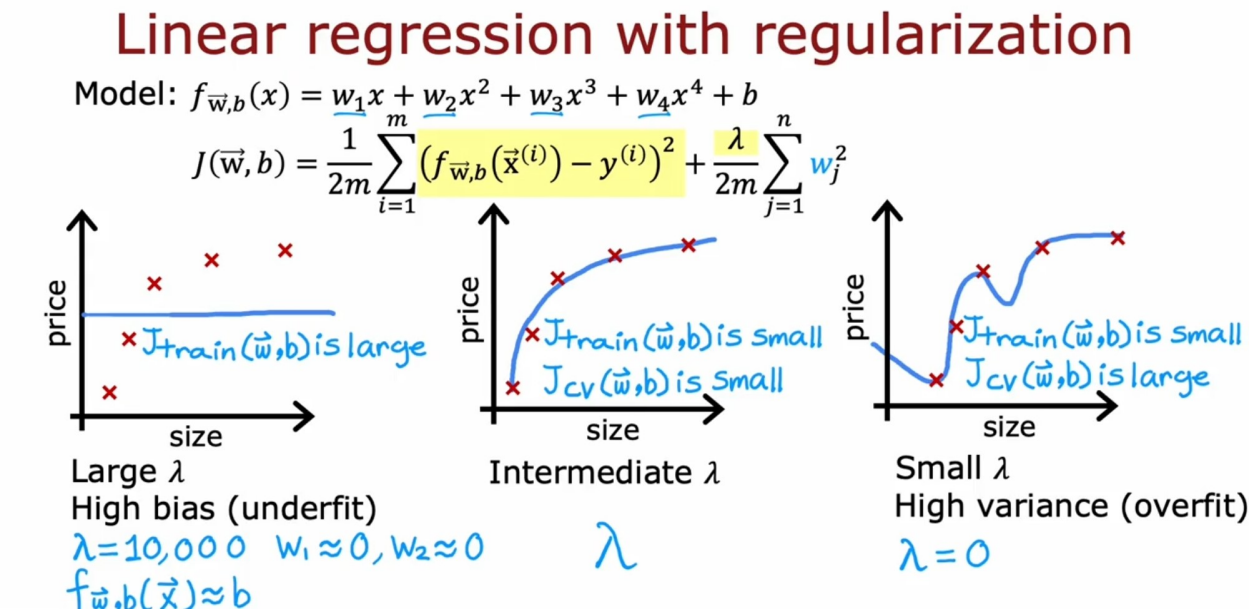
- High Bias and High Variance:

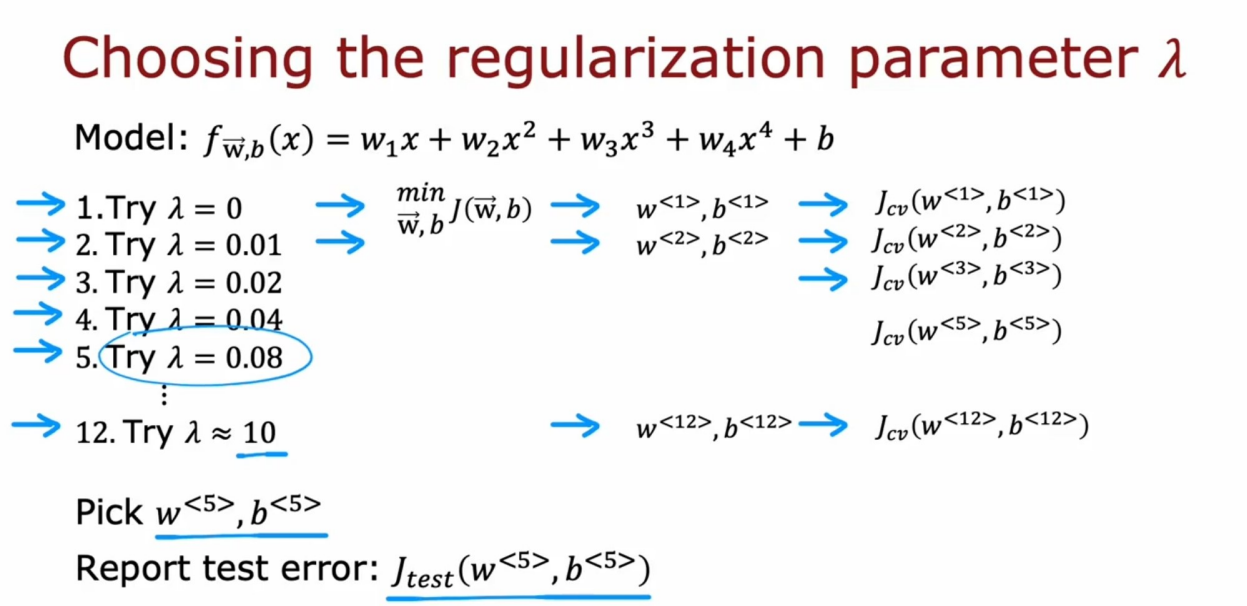
  $J_{\text{cv}} \gg J_{\text{train}}$ and J_{train} is high.



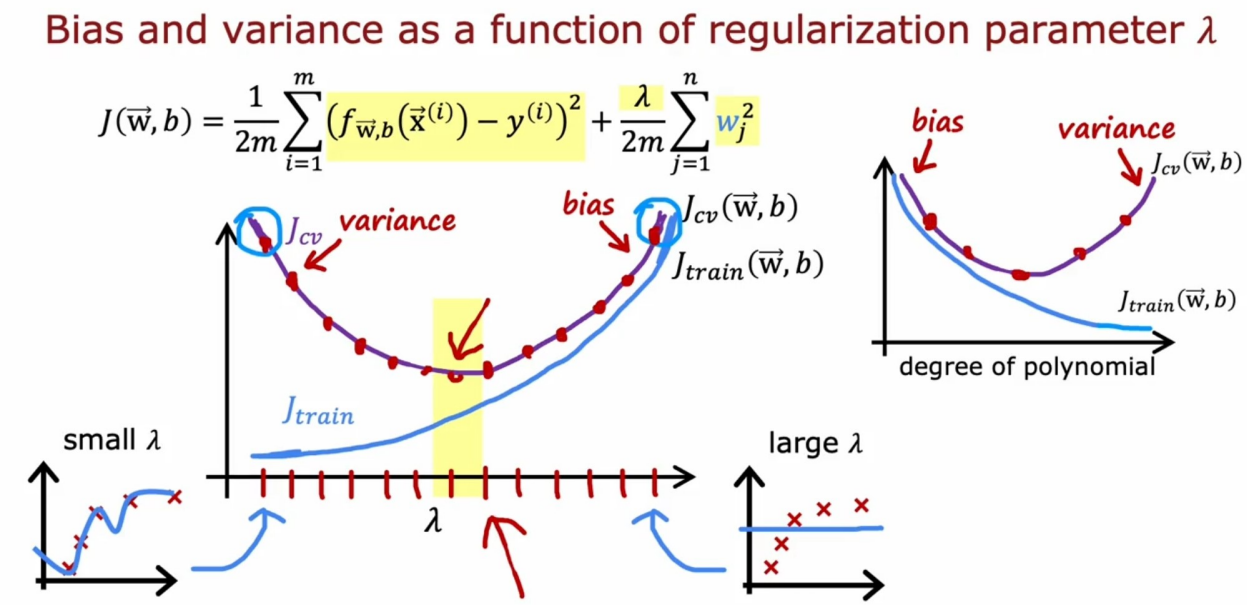
Regularization

We can use the error J_{train} and J_{cv} to choose the regularization parameter λ.





The two graphs are important. Notice the difference between the two, the first one's variable is λ, and the second one's variable is "degree of polynomial".



10.3 Baseline level of performance

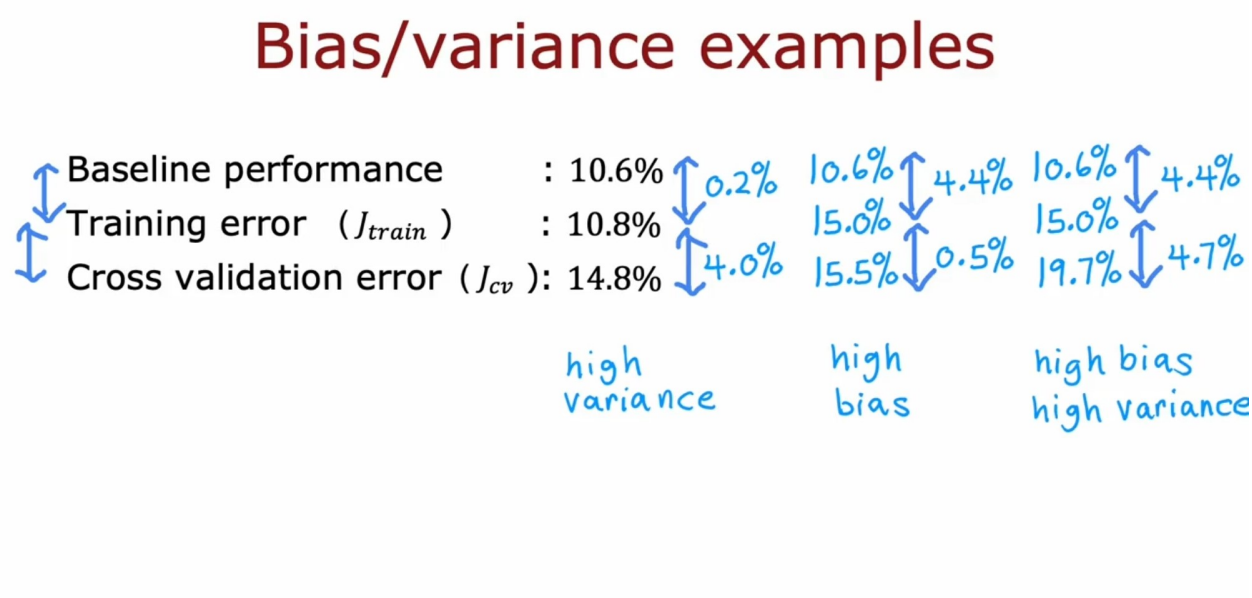
Establishing a baseline level of performance is important. It can be used to compare the performance of your model.

There are some ways to establish a baseline level of performance:

- Human-level performance: the error rate that a human can achieve.
- Competing algorithms performance
- Guess based on experience

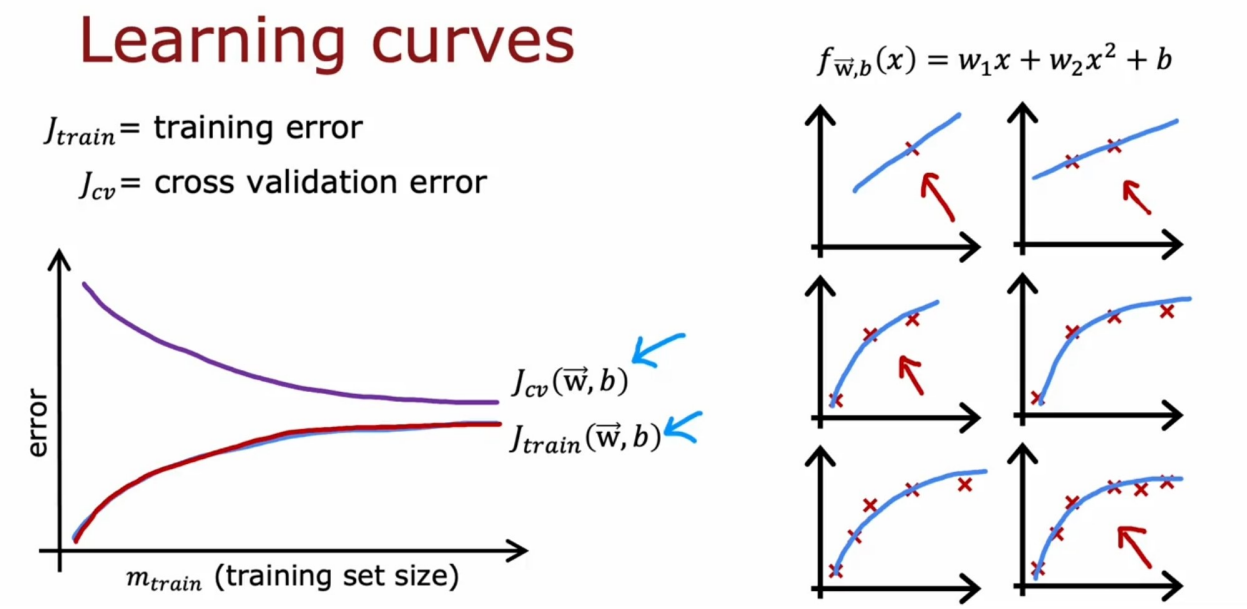
The baseline performance can be used to diagnose bias and variance.

- If the gap between the training error and the baseline performance is large, then the model may have high bias.
- If the gap beween the cross validation error and the training error is large, then the model may have high variance.



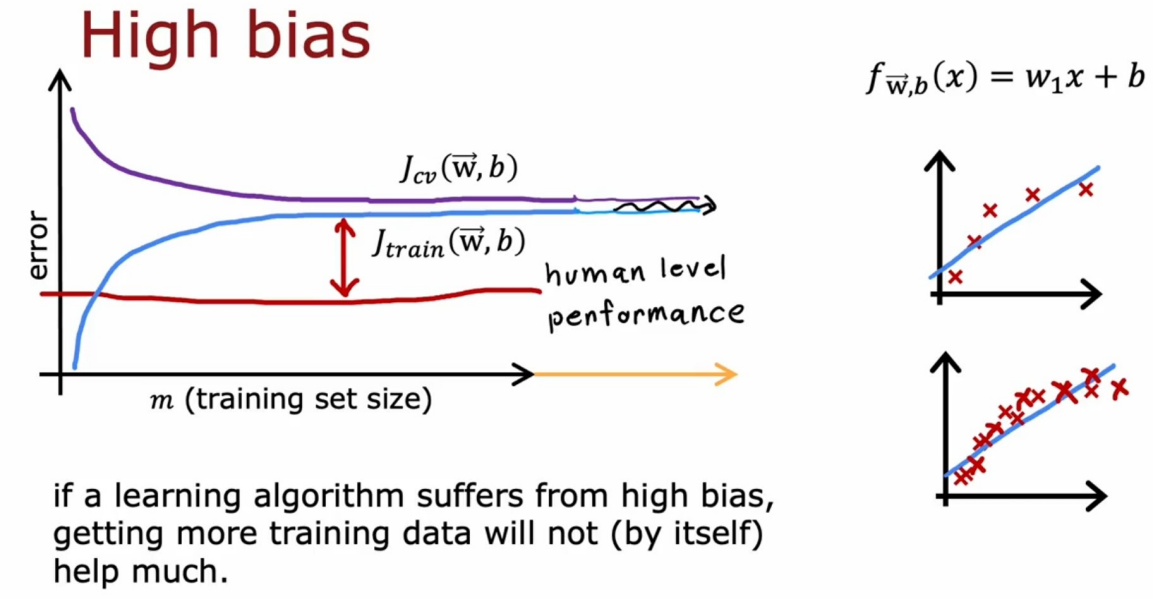
10.4 Learning curves

Usually, as the number of training examples increases, the cross-validation error will decrease while the training error will increase.



High bias

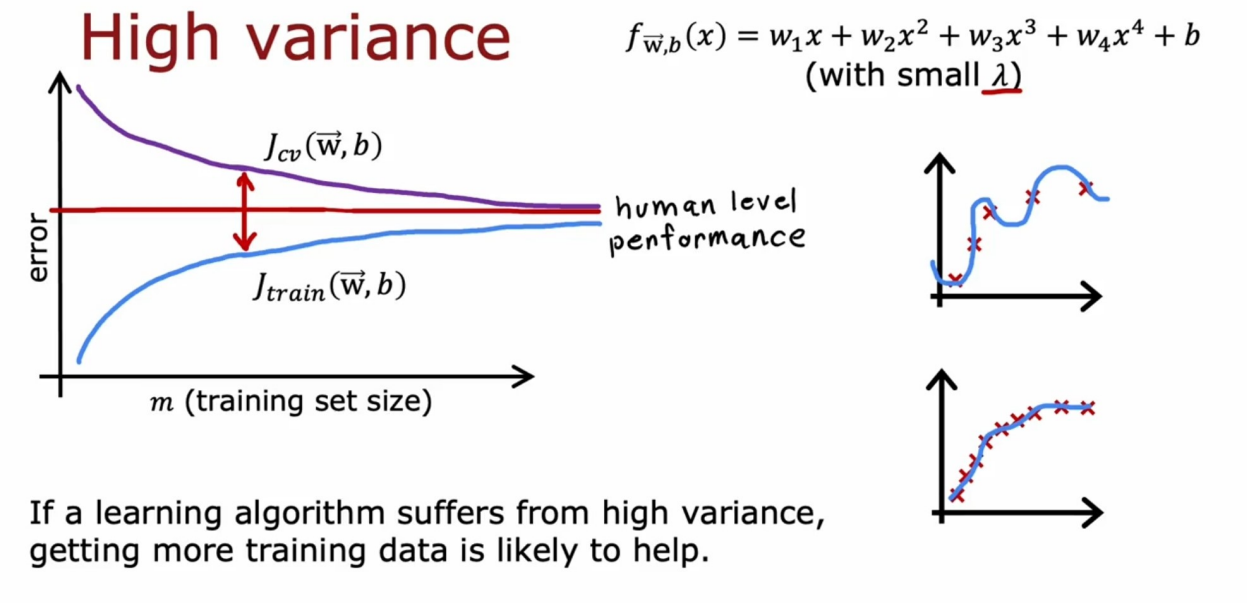
As the training data increases, J_cv and J_train will become flat, but still have a gap beween the baseline performance.



Because of that, getting more training data will not help much.

High variance

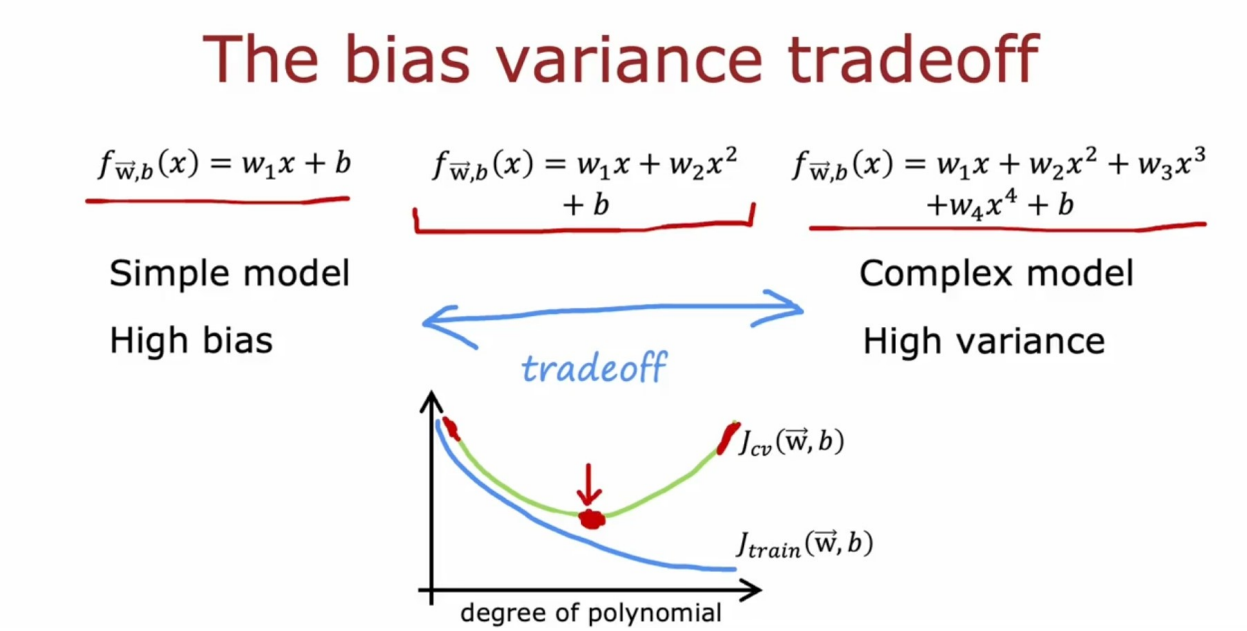
As the traing data increases, J_cv and J_train will become closer to the baseline performance.



High variance can be solved by lifting the size of training set, but this process may take a large amount of data.

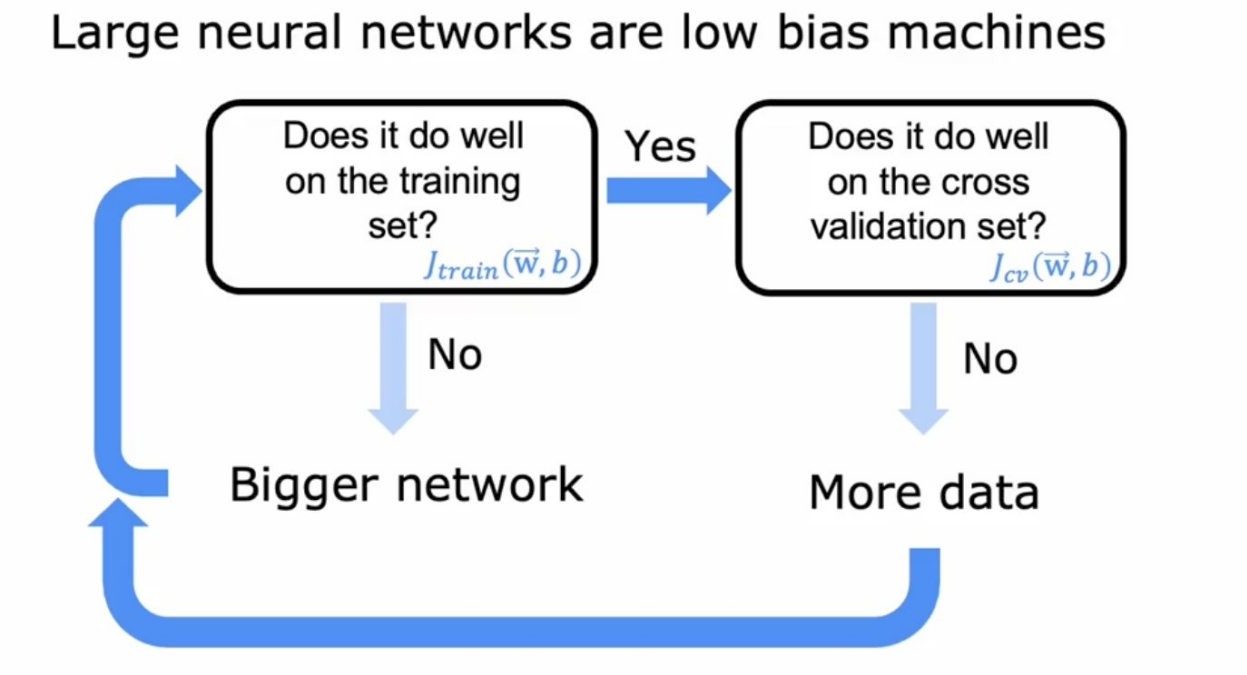
Debug a learning algorithm

- Get more training examples — fix high variance
- Try smaller sets of features — fix high variance
- Try getting additional features — fix high bias
- Try adding polynomial features($x_1^2, x_2^2, x_1x_2, \cdots$) — fix high bias
- Try decreasing λ — fix high bias
- Try increasing λ — fix high variance



Neural networks

A large neural network often has low bias, and it will usually do well or better than a smaller neural network as long as regularization is used to control overfitting.



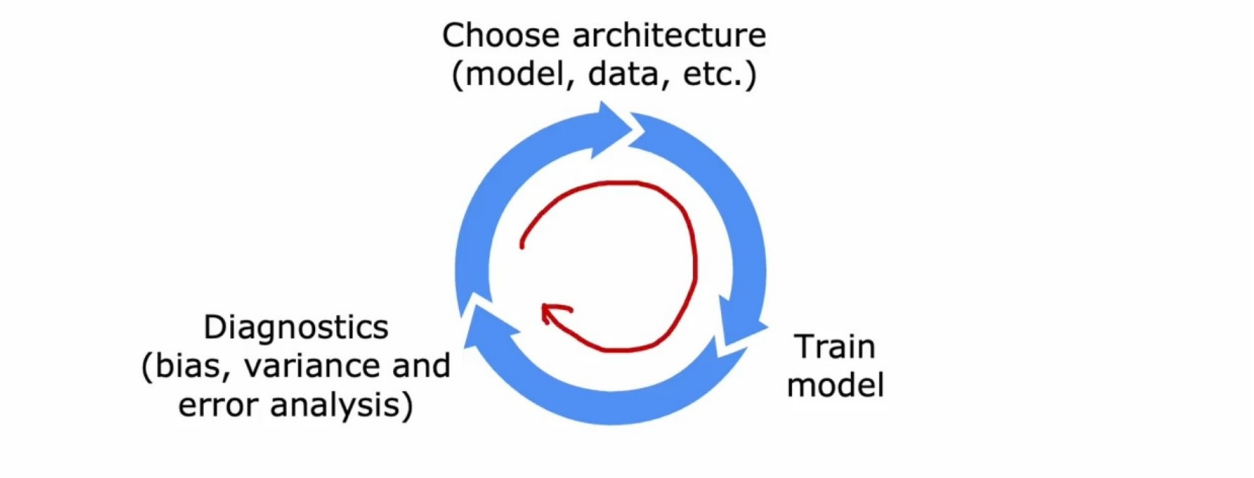

```
    layer_1 = Dense(units=25, activation='relu', kernel_regularizer=L2(0.01))
    layer_2 = Dense(units=15, activation='relu', kernel_regularizer=L2(0.01))
```

```
    layer_3 = Dense(units=1, activation='sigmoid', kernel_regularizer=L2(0.01))
    model = Sequential([layer_1, layer_2, layer_3])
```

The code above is an example of how to add regularization to a neural network. The `kernel_regularizer` parameter is used to add regularization to the weights of the layer.

10.5 Machine learning develop process

Iterative loop of machine learning

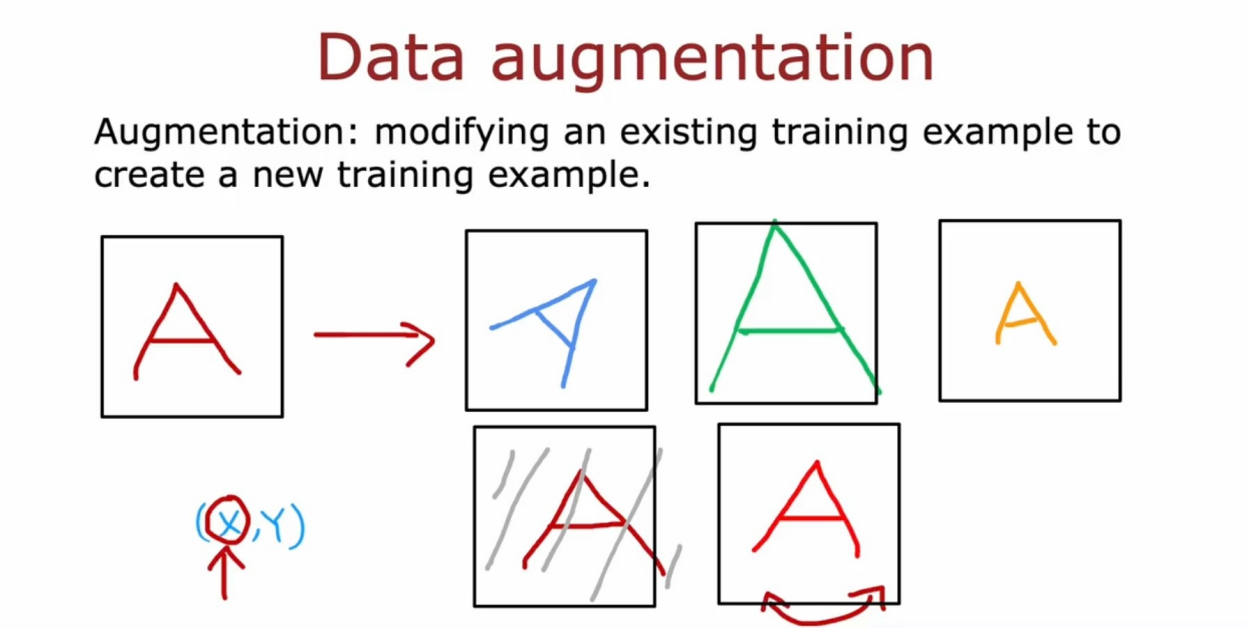


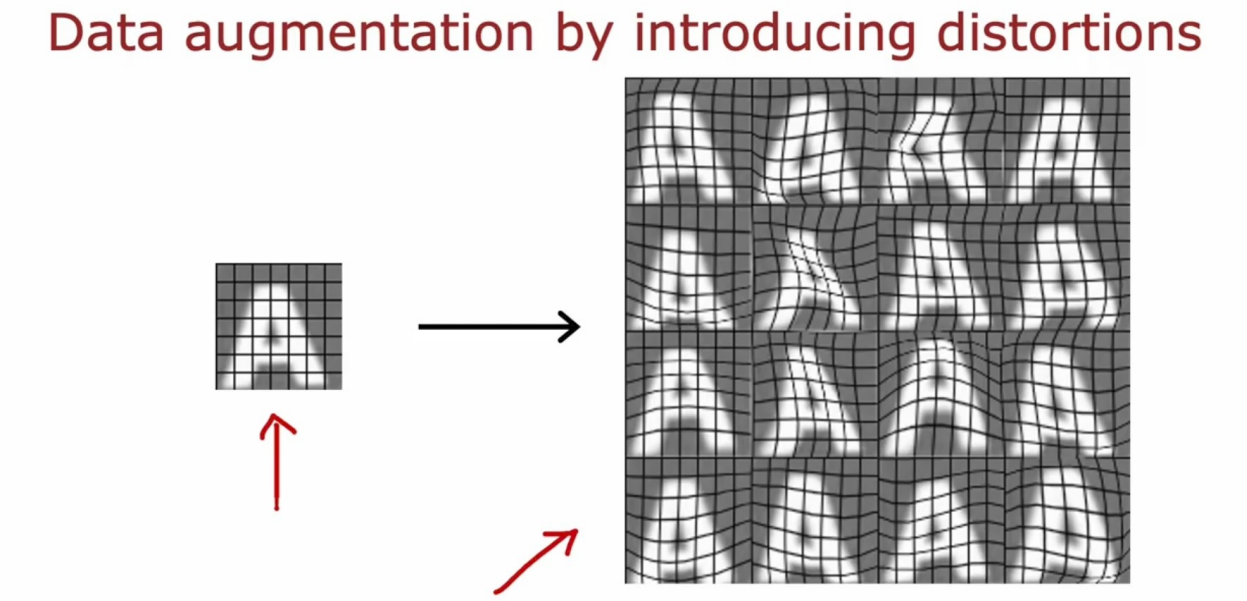
Error analysis

- Get a lot of ideas
- Use error analysis to prioritize ideas
- Error analysis means manually examining the examples in the cross-validation set that the algorithm made errors on.
- Error analysis can give you insights into what to do next.

Adding data

Data augmentation:





Data augmentation for speech

Speech recognition example

Original audio (voice search: "What is today's weather?")

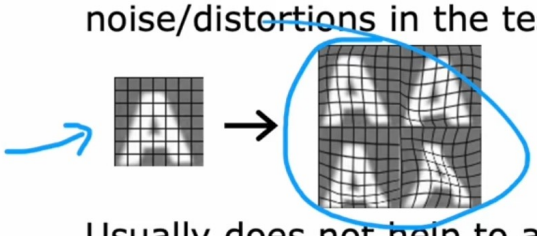
\+ Noisy background: Crowd

\+ Noisy background: Car

\+ Audio on bad cellphone connection

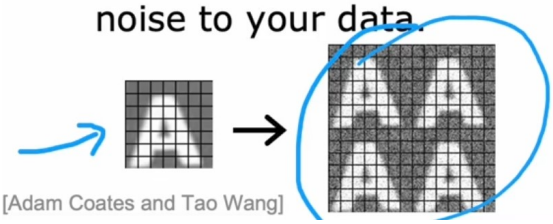
Data augmentation by introducing distortions

Distortion introduced should be representation of the type of noise/distortions in the test set.



Audio:
Background noise,
bad cellphone connection

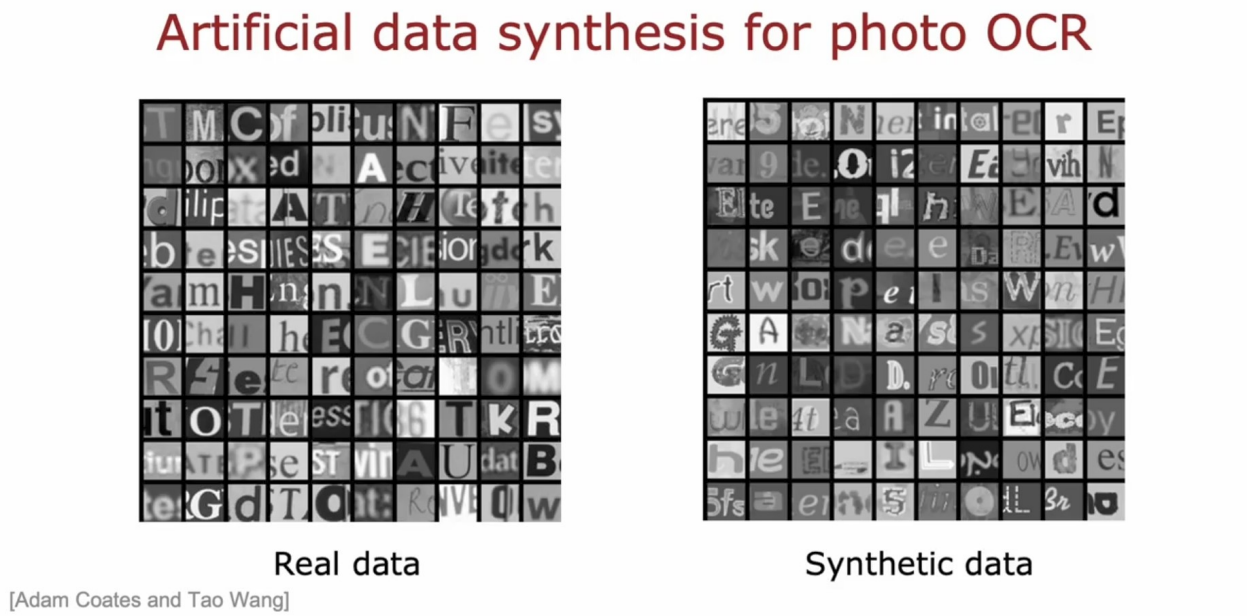
Usually does not help to add purely random/meaningless noise to your data.



[Adam Coates and Tao Wang]

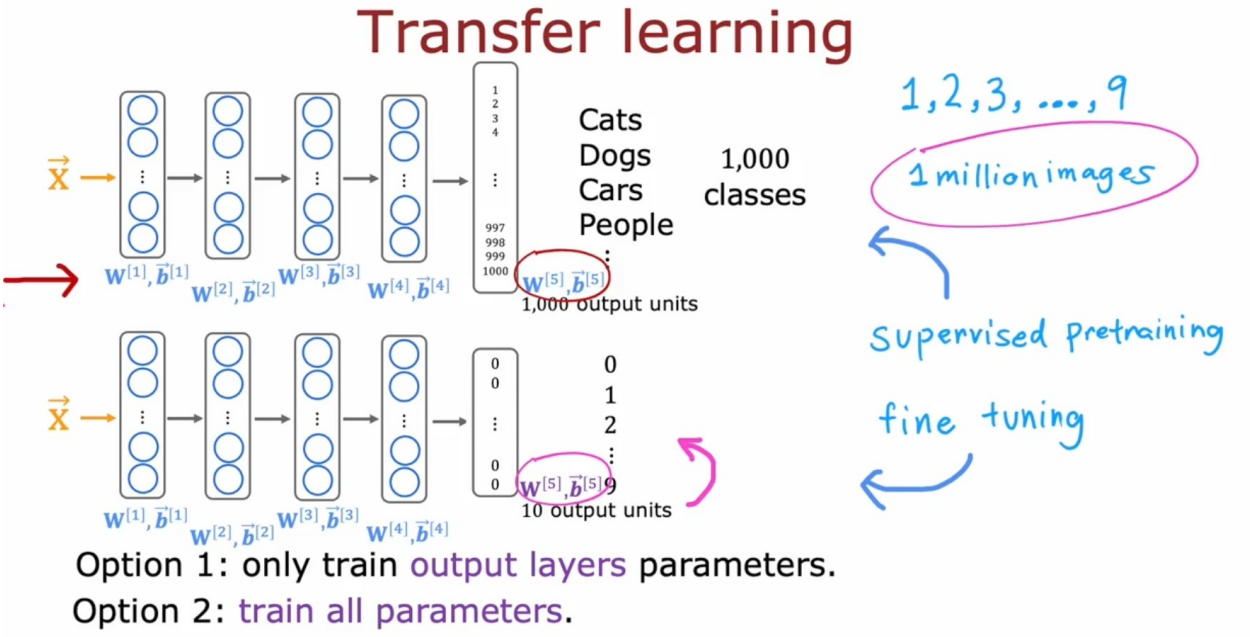
x_i=intensity (brightness) of pixel i

$x_i \leftarrow x_i$ + random noise



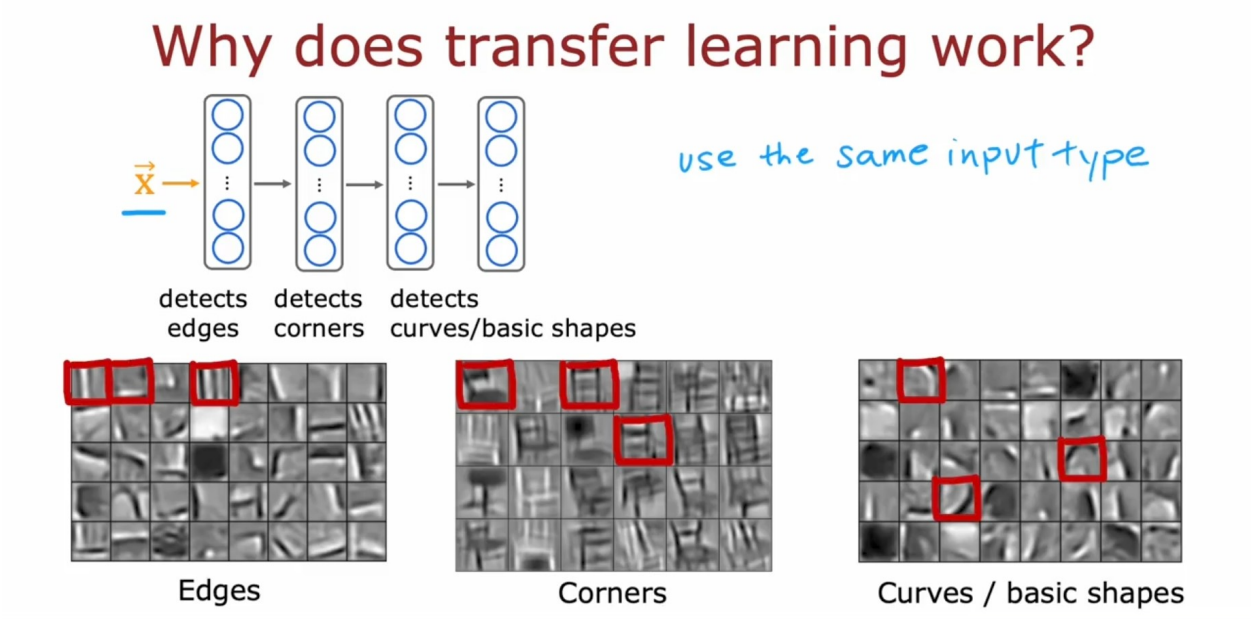
Transfer learning

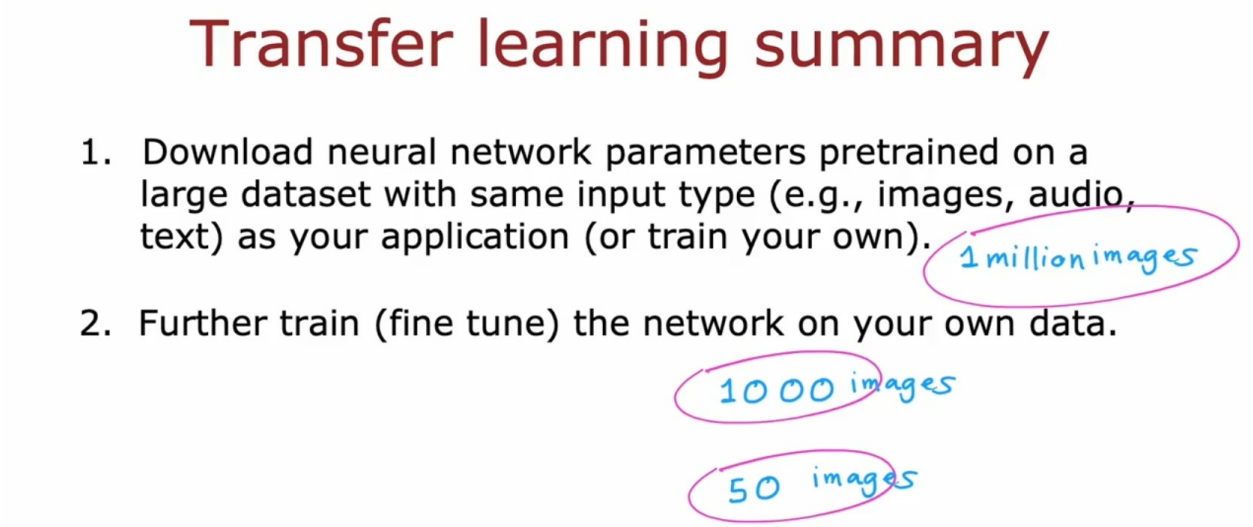
Transfer learning has 2 steps: **supervised pretraining** and **fine-tuning**. We can use a pre-trained model and fine-tune it to our problem. To do it, we can only change the output layer of the model.



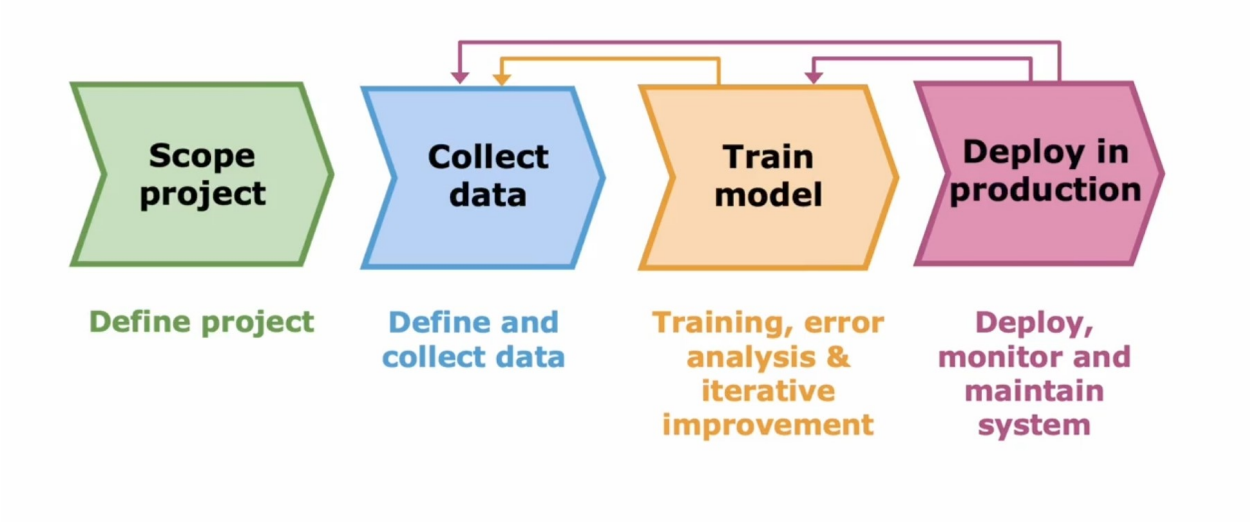
The reason for that is when input are of the same type, the first few layers of the neural network will learn features that are useful for the new task. Taking the CV model as an example, the first few layers will learn features that are general to all images such as edges, shapes,

etc.


Why does transfer learning work?
use the same input type
detects edges
detects corners
detects curves/basic shapes
Edges
Corners
Curves / basic shapes


Transfer learning summary
1. Download neural network parameters pretrained on a large dataset with same input type (e.g., images, audio, text) as your application (or train your own).
1 million images
2. Further train (fine tune) the network on your own data.
1000 images
50 images

Full cycle of machine learning project



10.6 Skewed datasets

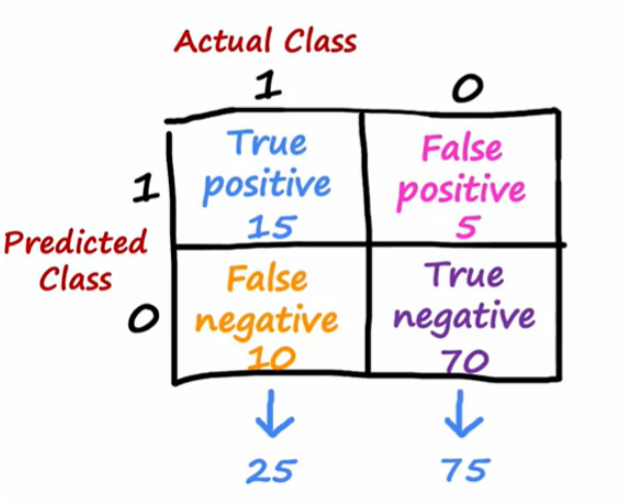
Error metrics for skewed datasets

Skewed datasets are datasets where the classes are not represented equally. For example, in a cancer dataset, the number of patients with cancer is much smaller than the number of patients without cancer.

In skewed datasets, accuracy is not a good metric to evaluate the model. For instance, if we have a dataset with 99% of the examples being negative and 1% being positive, a model that predicts all examples as negative will have an accuracy of 99%. While this model has a high accuracy, it is not useful because it does not predict any positive examples.

Therefore, we need to use other metrics to evaluate the model. We can divide the predictions into four categories:

- True positive (TP): the model correctly predicts the positive class.
- False positive (FP): the model incorrectly predicts the positive class.
- True negative (TN): the model correctly predicts the negative class.
- False negative (FN): the model incorrectly predicts the negative class.



There are 2 metrics that are commonly used to evaluate models on skewed datasets:

- Precision: the fraction of positive predictions that are correct.

$$\text{Precision} = \frac{\text{True positive}}{\text{total predicted positive}} = \frac{\text{True pos}}{\text{True pos} + \text{False pos}}$$

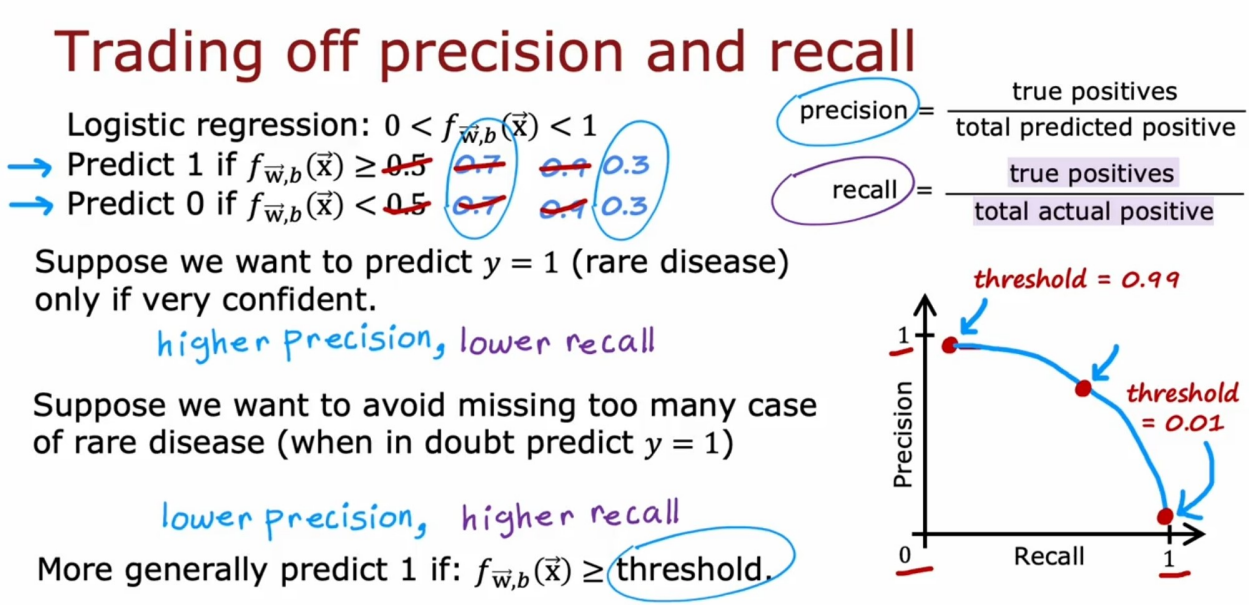
- Recall: the fraction of the positive examples that the model correctly predicts.

$$\text{Recall} = \frac{\text{True positive}}{\text{total actual positive}} = \frac{\text{True pos}}{\text{True pos} + \text{False pos}}$$

In this case, the “always negative” model has a precision of 0 and a recall of 0.(because the numerator of precision and recall is 0, note: the denominator of precision is also 0, but the precision and recall are defined as 0 in this case)

Trade off between precision and recall

There is a trade-off between precision and recall. We will set a threshold for predicting the positive class. If the probability of the positive class is greater than the threshold, we will predict the positive class, negative otherwise. If we increase the threshold for predicting the positive class, the precision will increase, but the recall will decrease.



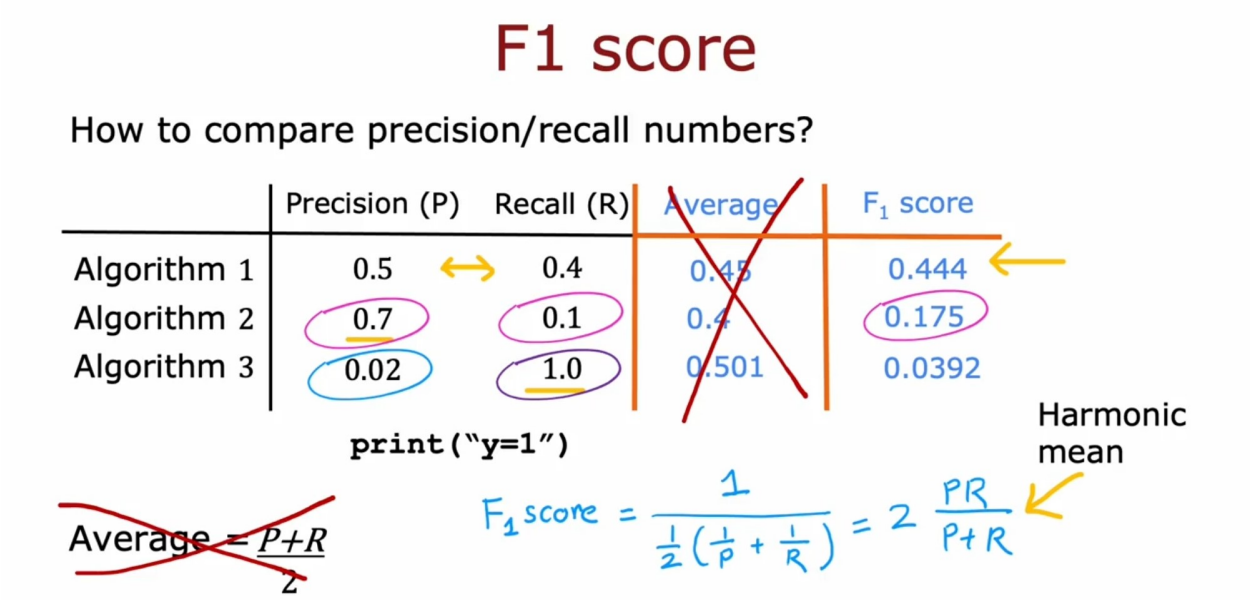
F1 score

The F1 score is the harmonic mean of precision and recall.

Theorem 10.6.1 ▸ F1 score

$$P := \text{Precision} \qquad R := \text{Recall}$$

$$\text{F}_1 \text{ score} = \frac{2}{\frac{1}{P} + \frac{1}{R}} = 2 \cdot \frac{P \cdot R}{P + R} \tag{10.1}$$

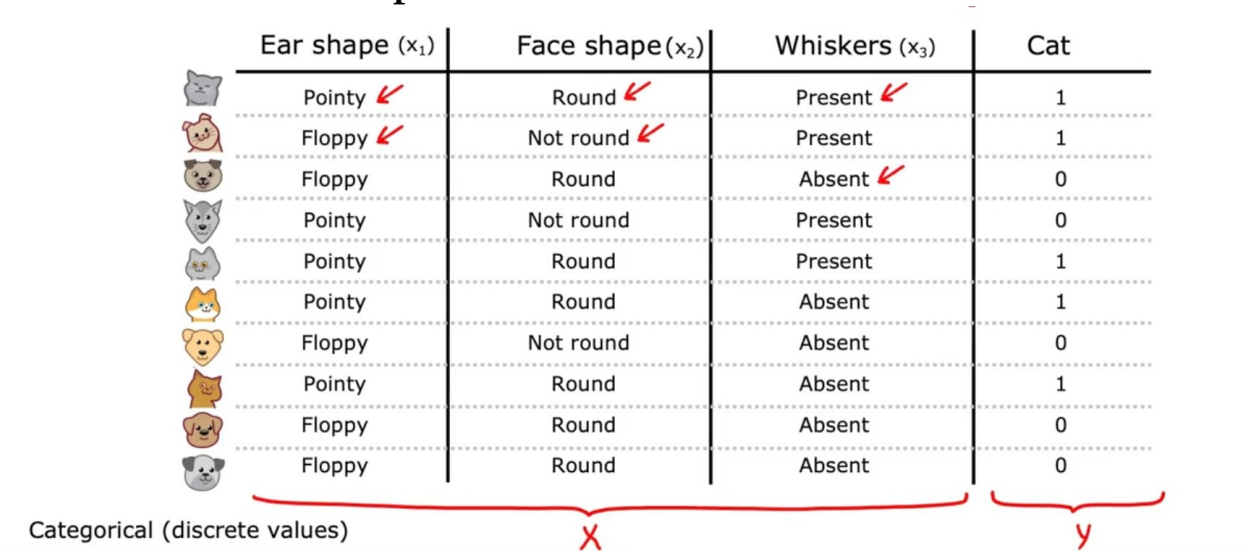

F1 score
How to compare precision/recall numbers?
Precision (P)
Recall (R)
Average
F1 score
Algorithm 1
0.5
0.4
0.45
0.444
Algorithm 2
0.7
0.1
0.4
0.175
Algorithm 3
0.02
1.0
0.501
0.0392
print("y=1")
Harmonic mean
Average = P+R / 2
F1 score = 1 / (1/2 (1/P + 1/R)) = 2 PR / P+R

11 Decision Trees

11.1 Introduction

Decision trees are a popular method for various machine learning tasks. They are easy to understand and interpret, and the process of building a decision tree is intuitive.

Cat classification example:



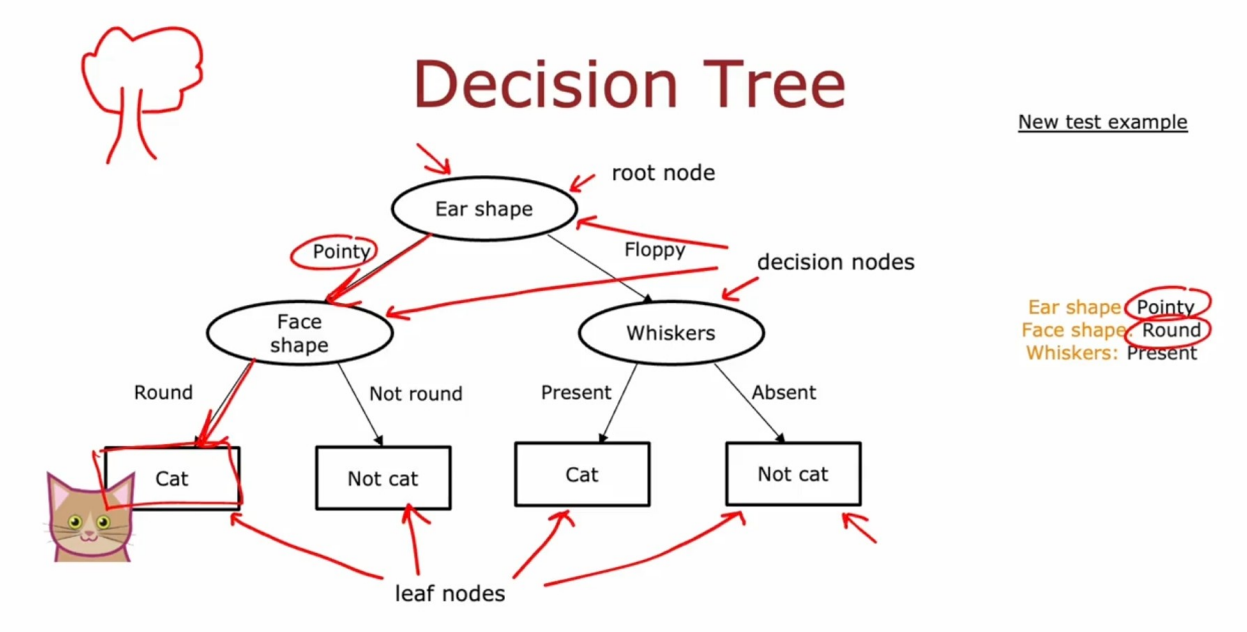
Ear shape (x_1)	Face shape (x_2)	Whiskers (x_3)	Cat
Pointy	Round	Present	1
Floppy	Not round	Present	1
Floppy	Round	Absent	0
Pointy	Not round	Present	0
Pointy	Round	Present	1
Pointy	Round	Absent	1
Floppy	Not round	Absent	0
Pointy	Round	Absent	1
Floppy	Round	Absent	0
Floppy	Round	Absent	0

Categorical (discrete values)

X

y

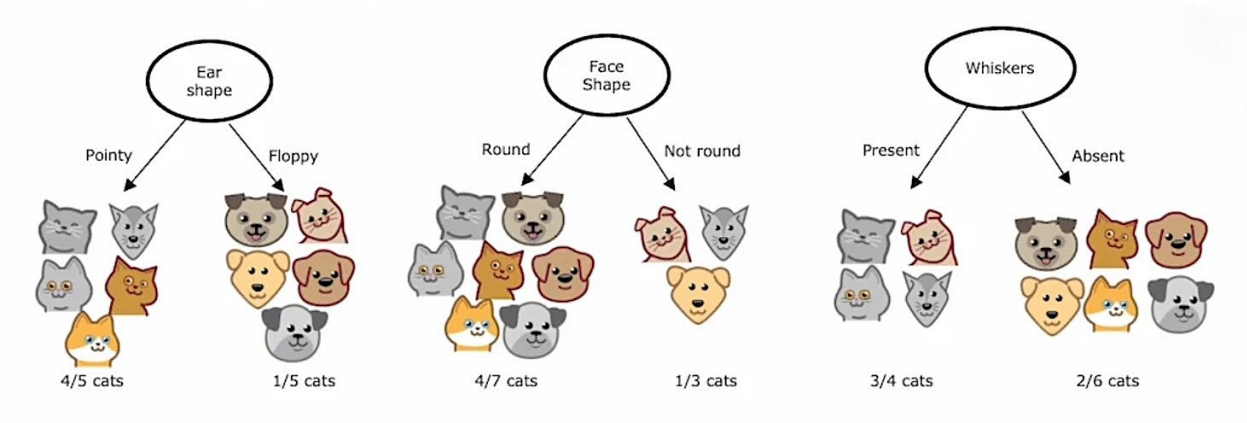
Decision tree model has the structure of a tree, where each **decision node** represents a feature (attribute), and each **leaf node** represents the outcome.



Learning process

The algorithm's goal is to find the decision tree that best classifies the training data. The learning process is divided into two steps:

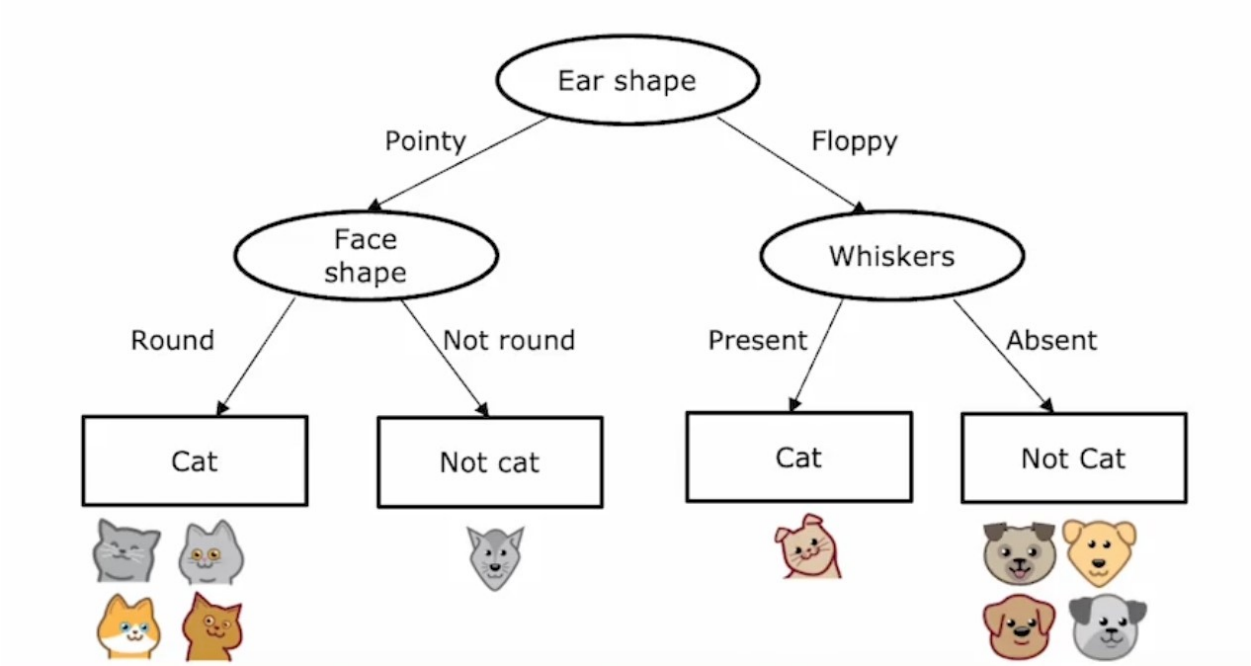
Decision 1: How to choose what features to split on at each node?
maximize purity(minimize impurity) of the child nodes.



Decision 2: When to stop splitting?

- When all data points in a node belong to the same class.
- When Exceeding a maximum depth.

- When impovements in purity are below a certain threshold.
- When the number of data points in a node is below a certain threshold.



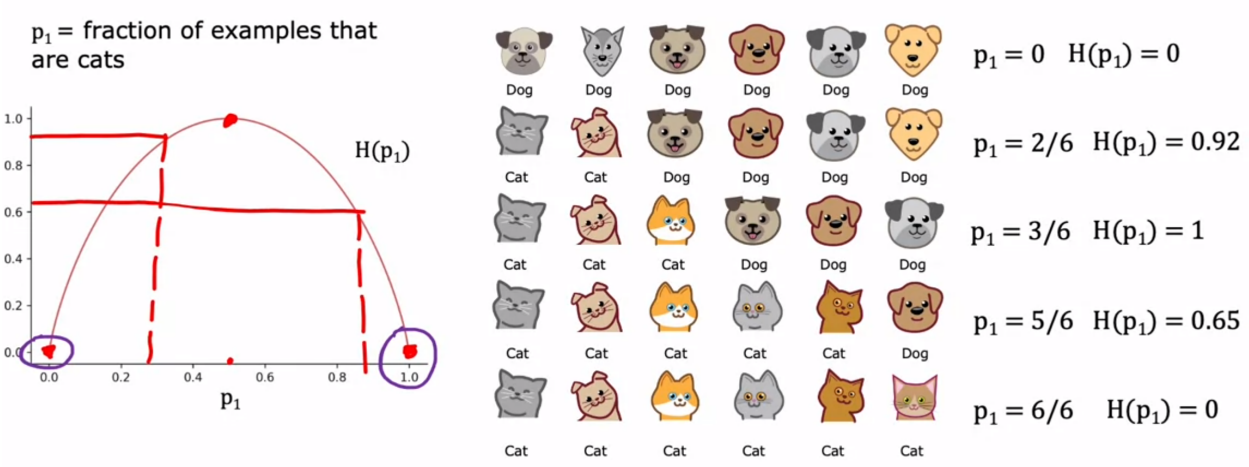
The splitting method and stopping method have a lots of variance, as the development of decision tree algorithm, many methods have been proposed.

11.2 Decision tree learning

Measuring purity

Entropy

Entropy is a measure of impurity.



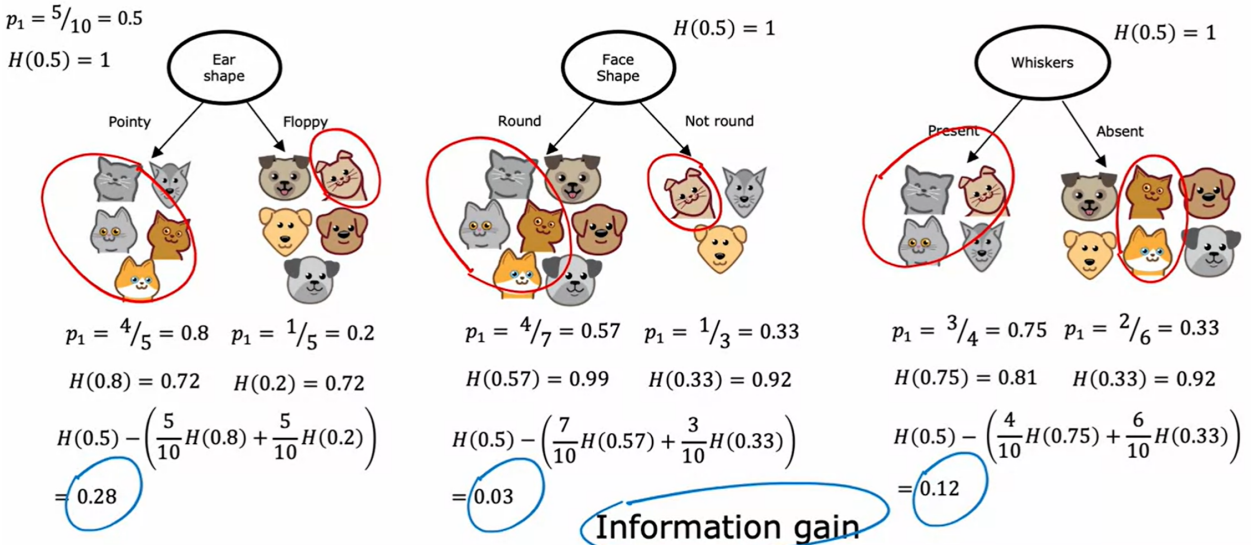
Theorem 11.2.1 ▸ Entropy

$$H(x) = -x \log_2(x) - (1-x)\log_2(1-x) \tag{11.1}$$

note: $0 \log_2(0) = 0$

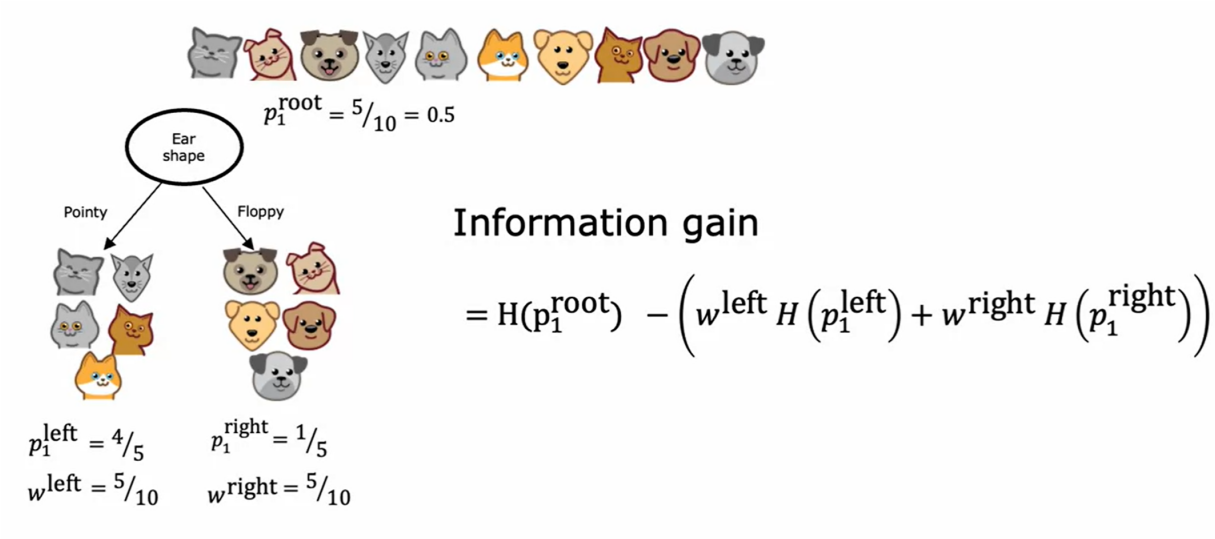
Information gain

Choosing splits that maximize information gain.



Theorem 11.2.2 ▸ Information gain

$$IG = H(p_1^{\text{parent}}) - \left(w^{\text{left}} \cdot H\left(p_1^{\text{left}}\right) + w^{\text{right}} \cdot H\left(p_1^{\text{right}}\right)\right) \tag{11.2}$$



Algorithm

Theorem 11.2.3 ▶ Decision tree algorithm

1.
2. `current node = root node`
3. `While not stopping criteria:`
   Calculate information gain for all features in current node
   Pick the feature with the highest information gain
   Split the node into child nodes according to the feature
   `current node = next(current node)`

This algorithm can be implemented recursively.

Theorem 11.2.4 ▶ Decision tree algorithm pseudocode

```
def build_decision_tree(node, data):
    if stopping_criterion(node, data):
        return
    feature, threshold = find_best_split(data)
    left_data, right_data = split_data(data, feature, threshold)
    node.left = build_decision_tree(node.left, left_data)
    node.right = build_decision_tree(node.right, right_data)
    return node
```

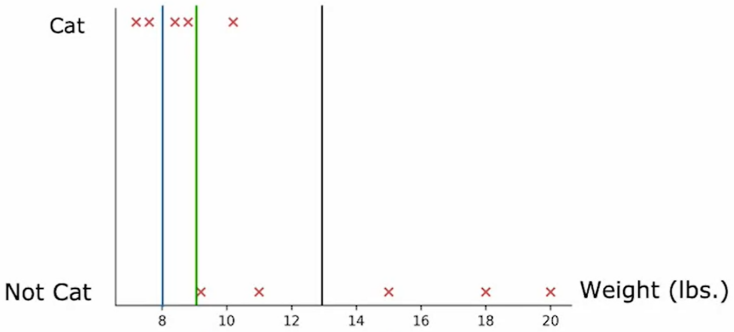
One-hot encoding

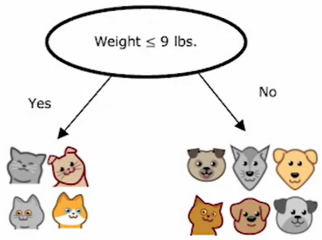
One-hot encoding is a method to convert categorical data into numerical data. if a feature has n categories, then it will be converted into n binary features.

	Pointy ears	Floppy ears	Round ears	Face shape	Whiskers	Cat
	1	0	0	~~Round~~ 1	~~Present~~ 1	1
	0	0	1	~~Not round~~ 0	~~Present~~ 1	1
	0	0	1	~~Round~~ 1	~~Absent~~ 0	0
	1	0	0	~~Not round~~ 0	~~Present~~ 1	0
	0	0	1	~~Round~~ 1	~~Present~~ 1	1
	1	0	0	~~Round~~ 1	~~Absent~~ 0	1
	0	1	0	~~Not round~~ 0	~~Absent~~ 0	1
	0	0	1	~~Round~~ 1	~~Absent~~ 0	1
	0	1	0	~~Round~~ 1	~~Absent~~ 0	1
	0	1	0	~~Round~~ 1	~~Absent~~ 0	1

Splitting continuous variables

Choose the best split point by calculating information gain for all possible split points.





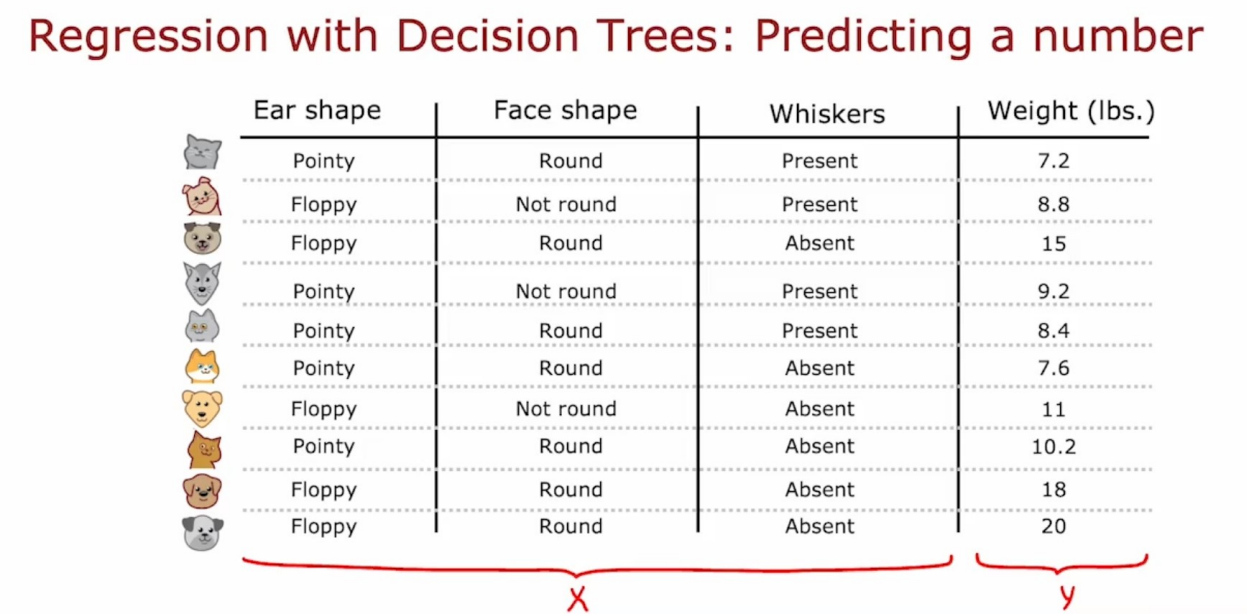
$$H(0.5) - \left(\frac{2}{10}H\left(\frac{2}{2}\right) + \frac{8}{10}H\left(\frac{3}{8}\right)\right) = 0.24$$

$$H(0.5) - \left(\frac{4}{10}H\left(\frac{4}{4}\right) + \frac{6}{10}H\left(\frac{1}{6}\right)\right) = 0.61$$

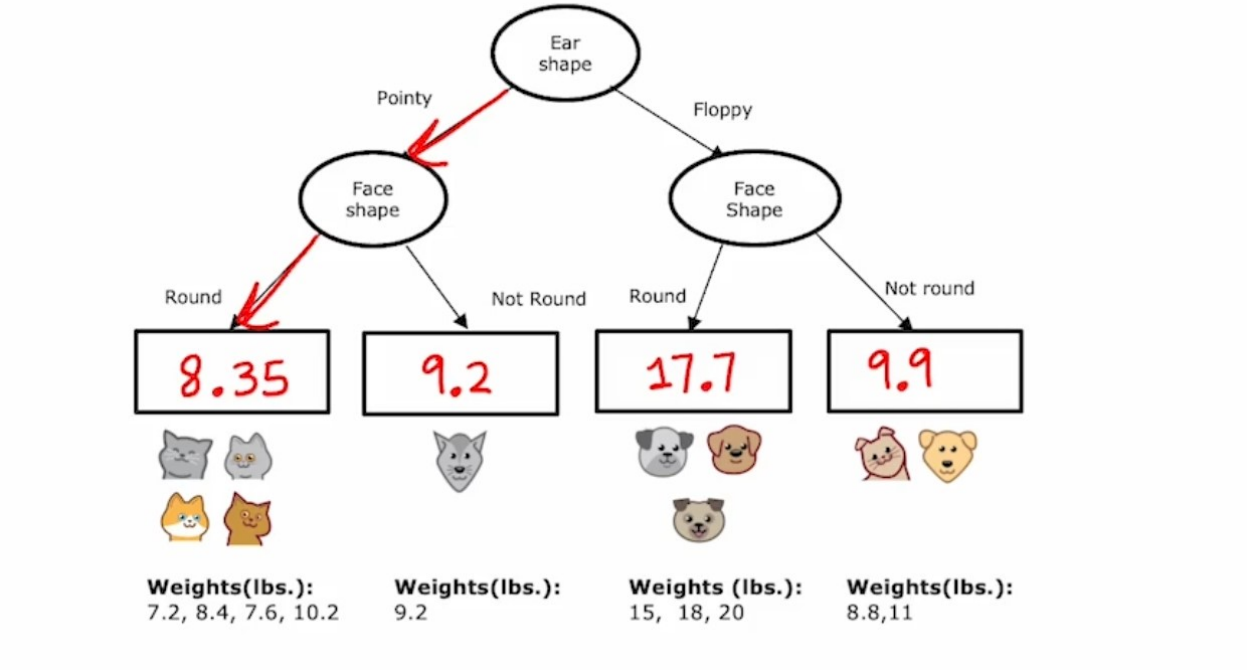
$$H(0.5) - \left(\frac{7}{10}H\left(\frac{5}{7}\right) + \frac{3}{10}H\left(\frac{0}{3}\right)\right) = 0.40$$

Regression trees

Regression trees are used for continuous target variables.

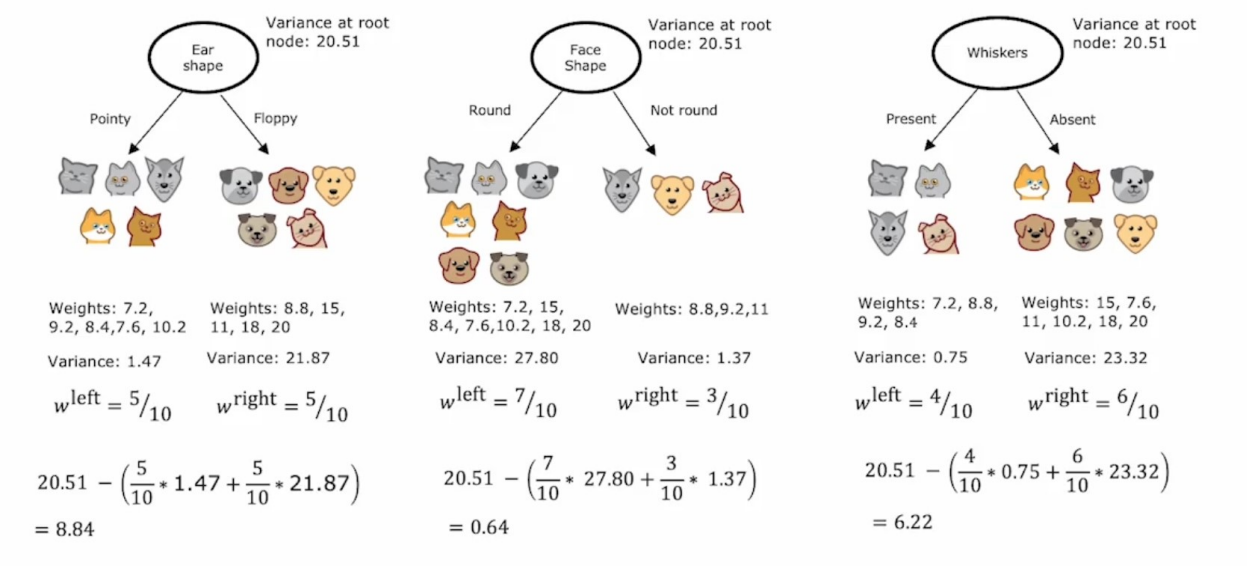


Assume that we have built a decision tree model, and we want to predict the target value for a new data point. The prediction is made by traversing the tree from the root node to a leaf node. The target value of the leaf node (average of target values of the training data in that node) is the prediction.



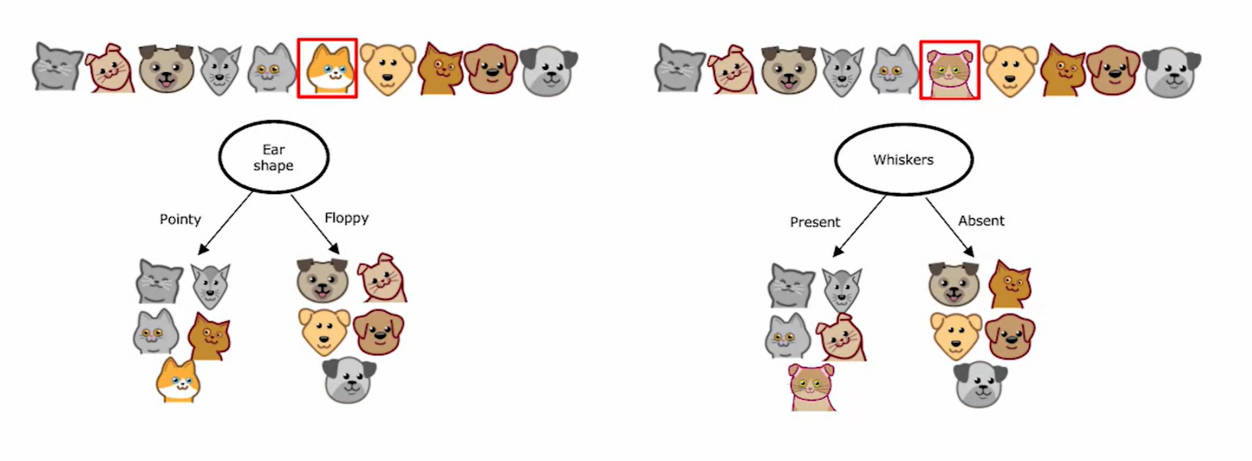
Choosing a split:

- Calculate the variance of the target values in the parent node.
- Calculate the variance of the target values in the child nodes.
- Calculate the weighted average of the child nodes' variances.
- Calculate the reduction in variance.
- Choose the split that maximizes the reduction in variance.

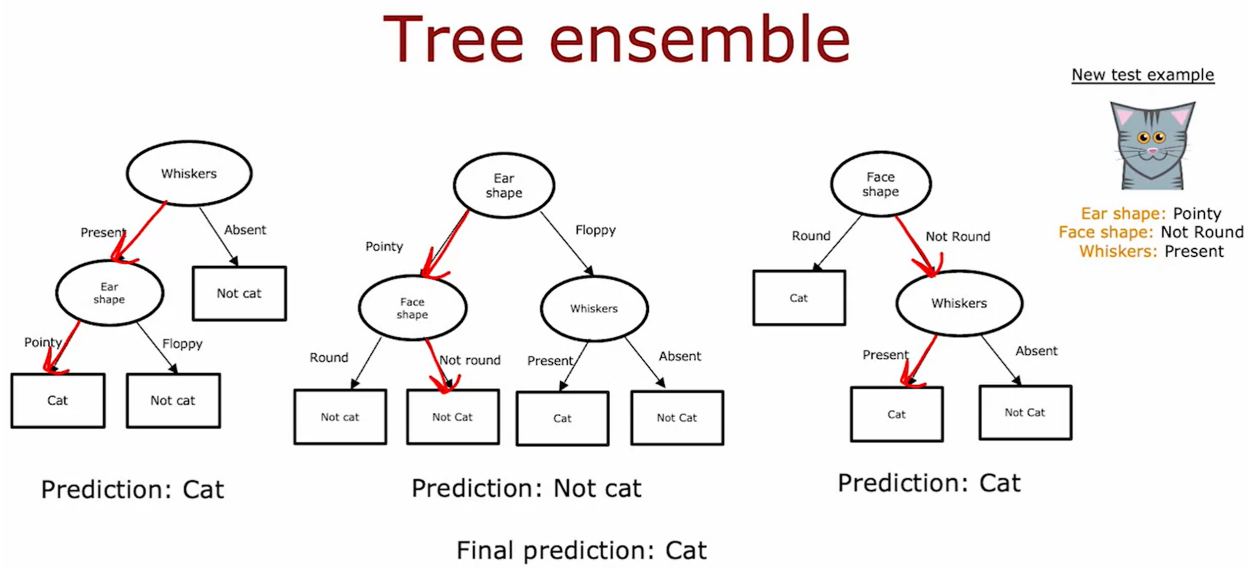


11.3 Tree ensembles

The downside of a single decision tree is that it is sensitive to the change in data, which is to say a small change in the training data can lead to a completely different tree. And this makes the model not robust.



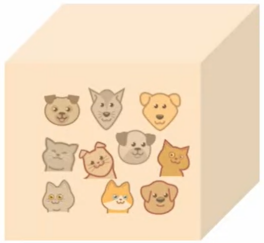
Therefore, we can use tree ensembles to improve the model's performance. Tree ensembles are a collection of decision trees that work together to make predictions. The process is that each tree makes a prediction, and the final prediction is the most voted prediction (or the average of all trees' predictions for numerical output).



Random forests and **boosted trees** are two popular tree ensemble methods. next we will first introduce random forests.

Sampling with replacement

"Replacement" means that after sample a data point, we will put it back to the dataset before sample the next data point. So the same data point can be sampled multiple times and not all the data points will be sampled.



Ear shape	Face shape	Whiskers	Cat
Pointy	Round	Present	1
Floppy	Not round	Absent	0
Pointy	Round	Absent	1
Pointy	Not round	Present	0
Floppy	Not round	Absent	0
Pointy	Round	Absent	1
Pointy	Round	Present	1
Floppy	Not round	Present	1
Floppy	Round	Absent	0
Pointy	Round	Absent	1

Bagged decision tree

Given training set of size m, we want to sample B groups of data points, each of size m. Then we build a decision tree for each group. This is called a bagged decision tree.

```
for b = 1 to B:
```

- Sample m data points with replacement to create a training set.
- Train a decision tree on this new training set.

Randomizing the features

At each node, when choosing a feature to split on, if n features are available, we can choose a subset of k features and allow the algorithm to only choose from this subset.

By using a subset of features to split at each node, we can reduce the correlation between trees.

> A common choice for the parameter k is $\sqrt{n}$.

Random forests

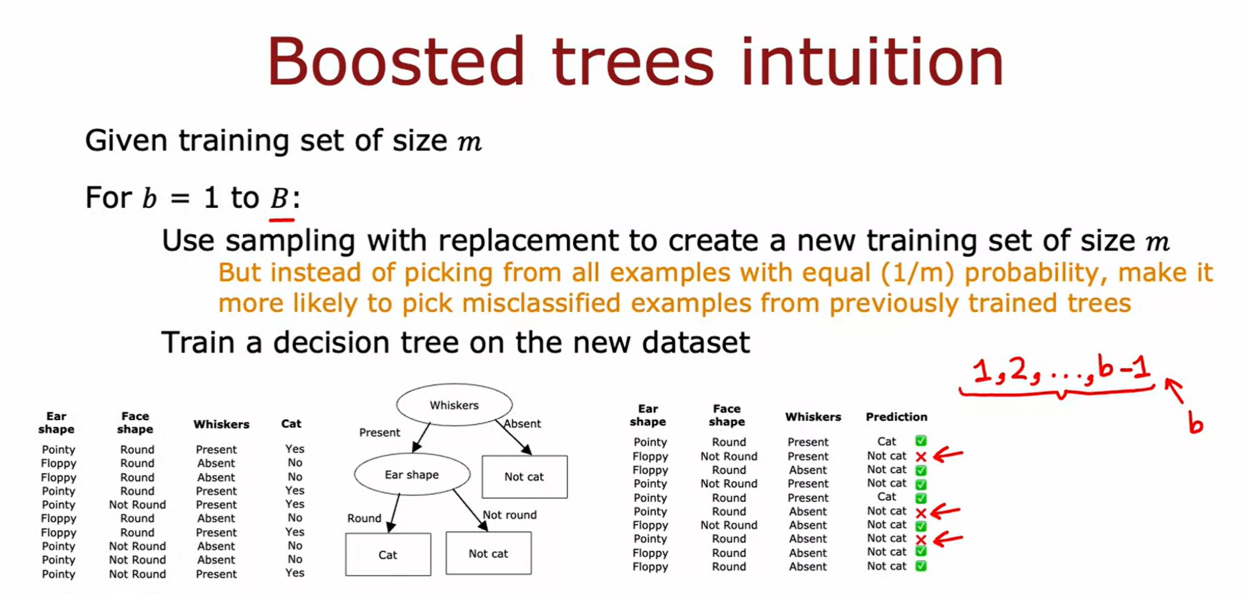
So the random forest algorithm is a bagged decision tree with randomizing the features.

Random forests will work typically much better and becomes much more robust than just a single decision tree. One way to think about why this is more robust than a single decision tree is the sampling with replacement procedure causes the algorithm to explore a lot of small

changes to the data already and it's training different decision trees and is averaging over all of those changes to the data that the sampling with replacement procedure causes. And so this means that any little change further to the training set makes it less likely to have a huge impact on the overall output of the overall random forest algorithm. Because it's already explored and it's averaging over a lot of small changes to the training set.

Boosted trees

Boosted trees shows the idea of "deliberate training", this is to say rather than sampling with equal probability, we prefer to sample the data points that are hard to classify.



XGBoost

XGBoost (Extreme Gradient Boosting) is a popular implementation of the boosted tree algorithm. It has a lot of advantages:

- Open source implementation
- Fast efficient implementation
- Good choice of default splitting criteria and criteria for stop splitting
- Built in regularization to prevent overfitting
- Highly competitive algorithm for machine learning competitions(eg: Kaggle)

Implementation:

```
import xgboost as xgb

# for classification problem
model = xgb.XGBClassifier()
# for regression problem
model = xgb.XGBRegressor()

model.fit(X_train, y_train)
model.predict(X_test)
```

Decision trees vs. Neural networks

Decision trees and Tree ensembles:

- Works well on tabular (structured) data
- Not recommended for unstructured data (images, text, audio etc.)
- Fast training and prediction
- Small decision trees may be human interpretable and can be visualized

Neural networks:

- Works well on all types of data, including unstructured data and tabular data
- Slower training and prediction
- Works well with transfer learning
- When building a system of multiple models working together, it might be easier to string together multiple neural networks

Part III

Unsupervised Learning, Recommender Systems and Reinforcement Learning

Index

Definitions

0.2.1 Supervised Learning 3
0.2.2 Unsupervised Learning 3
0.2.3 Reinforcement Learning 3
2.1.1 Cost Function 3
3.3.2 Batch Gradient Descent 11
4.2.1 Hypothesis Function 12
4.4.1 Cost Function for Multiple Features 14
5.5.1 Polynomial regression 25
6.0.1 Classification 27
6.0.2 Binary classification 27
6.2.1 Logistic Regression 28
6.3.1 Logistic loss function 31
6.3.2 Logistic cost function 31
7.1.1 overfitting 34
9.3.1 Linear Activation 48
9.3.2 Sigmoid Activation 48
9.3.3 ReLU Activation 48
10.2.1 Bias 64
10.2.2 Variance 64

Examples

3.2.2 local minima 9
4.1.1 Notation for mutiple features . 12
5.5.2 Polynomial regression 25
6.1.1 Linear Regression for Binary Classification 27

Theorems

3.2.1 Gradient Descent Algorithm . 8
3.3.1 imporved Gradient Descent Algorithm 10
4.4.2 Gradient Descent for Multiple Features 14
5.2.1 mean normalization 21
5.2.2 z-score normalization 22
6.2.2 Logistic Regression 29
6.4.1 Gradient Descent 32
7.3.1 regularization cost function . . 36
7.3.2 regularization gradient descent 36
7.3.3 regularization cost function . . 37
7.3.4 regularization gradient descent 37
10.6.1 F1 score 78
11.2.1 Entropy 83
11.2.2 Information gain 84
11.2.3 Decision tree algorithm 85
11.2.4 Decision tree algorithm pseudocode 85

Code Snippets

2.2.1 Cost Function 3