Notes of Machine Learning Specialization

FlowerMouse

August 4, 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

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.

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

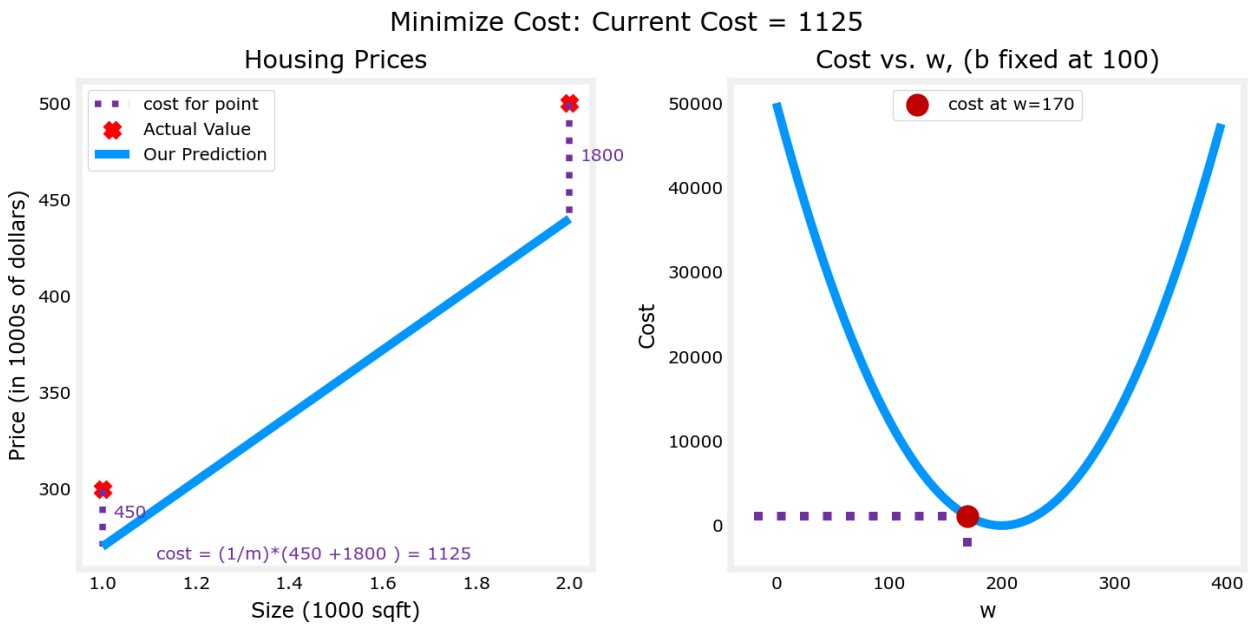
explaination the $\hat{y}^{(i)} - y^{(i)}$shows the distance of the predicted value from the actual value.

2.2 Cost Function Intuition

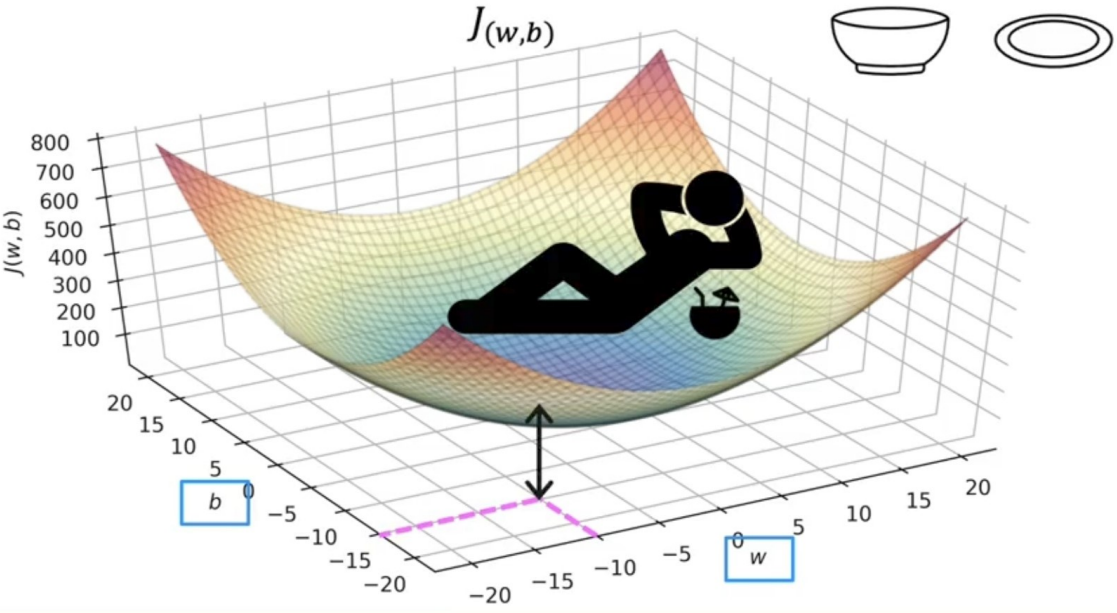
$J(w)$ with only one parameter w

Code Snippet 2.2.1 ▶ Cost Function

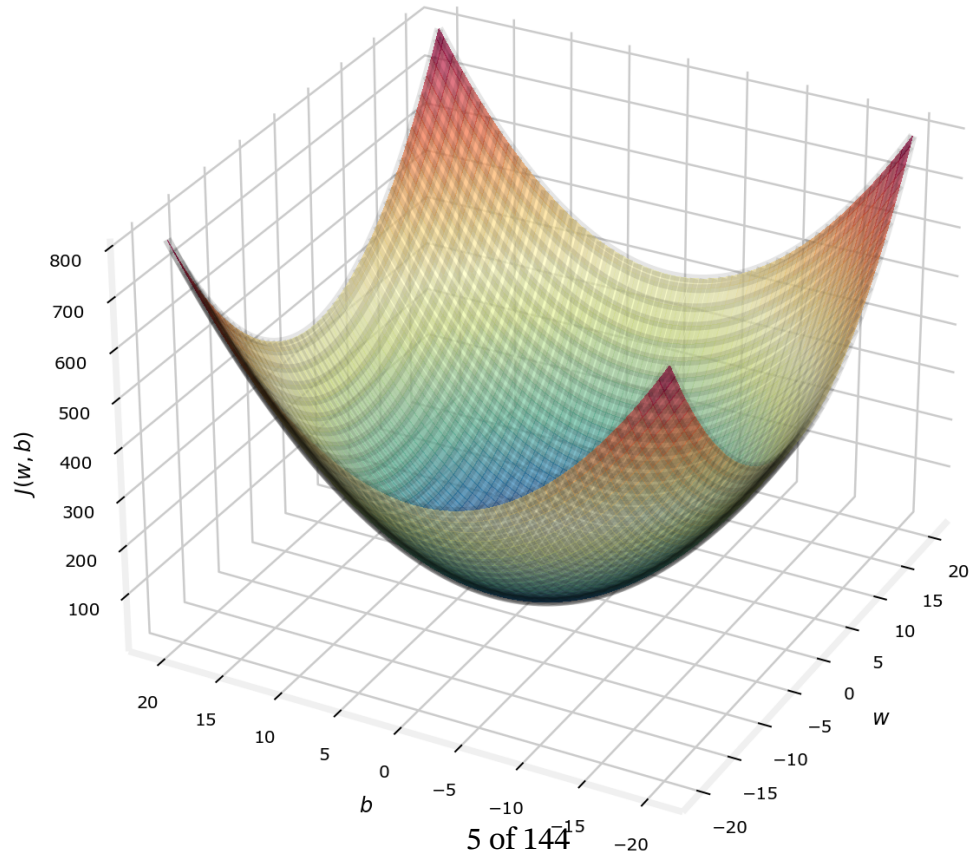
```
x_train = np.array([1.0, 2.0])           #(size in 1000 square feet)
y_train = np.array([300.0, 500.0])       #(price in 1000s of dollars)
plt_intuition(x_train,y_train)
```


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

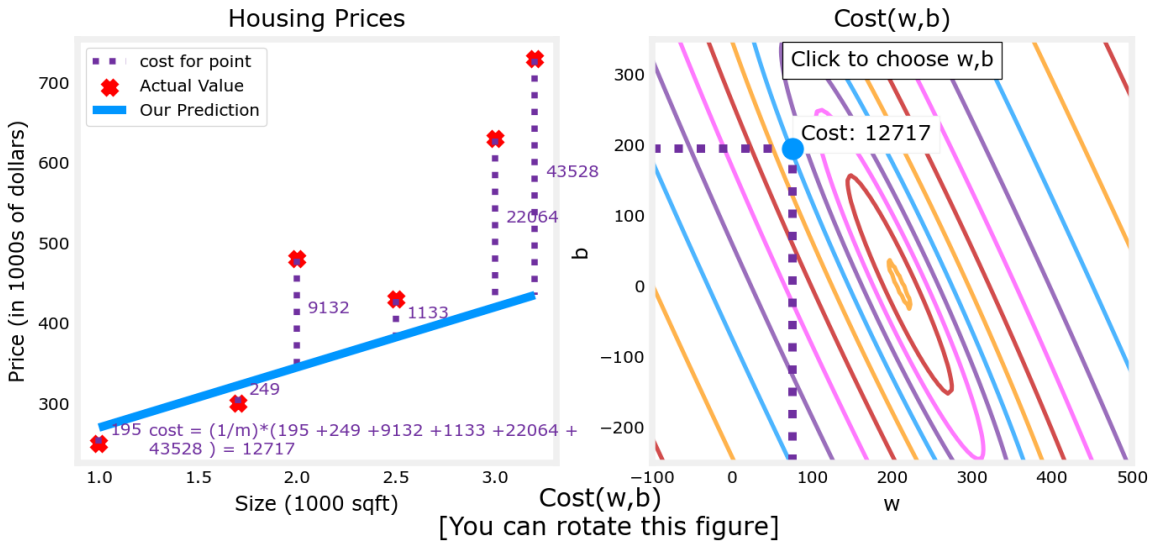
$J(w, b)$ with two parameters w and b

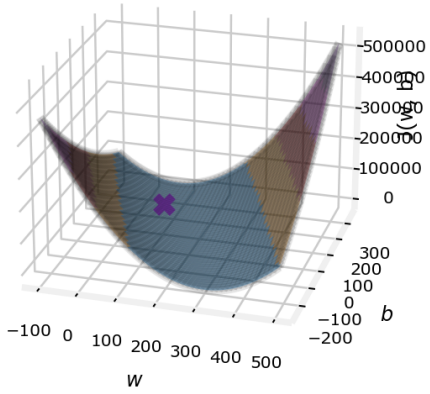


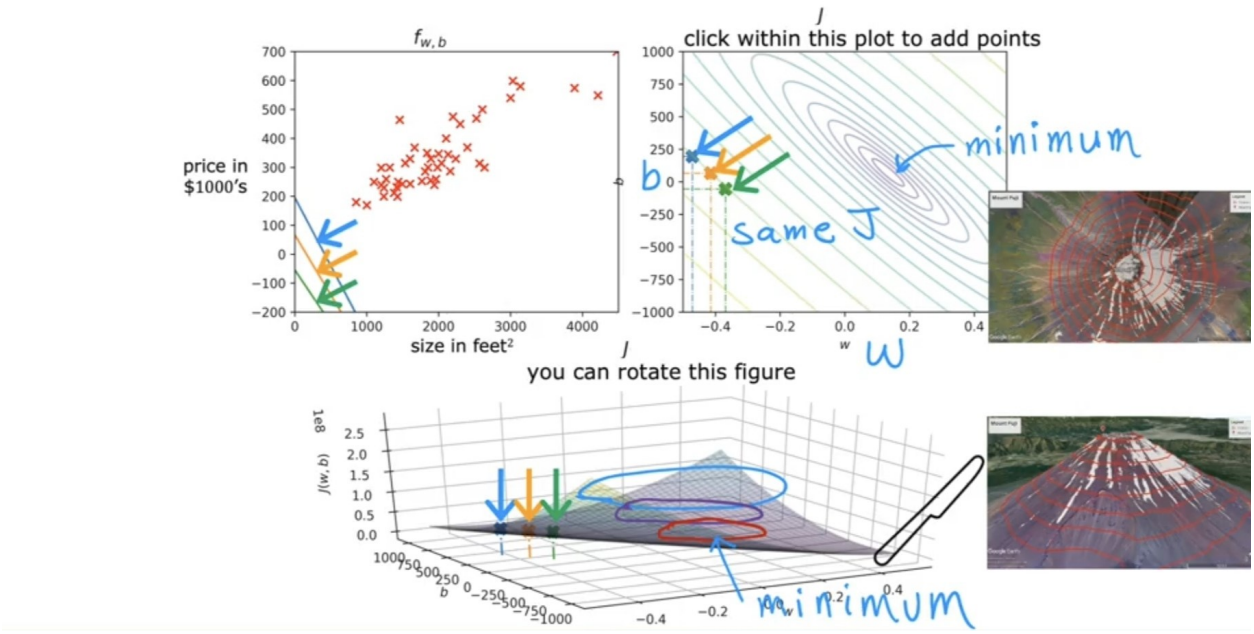
$J(w, b)$
[You can rotate this figure]



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



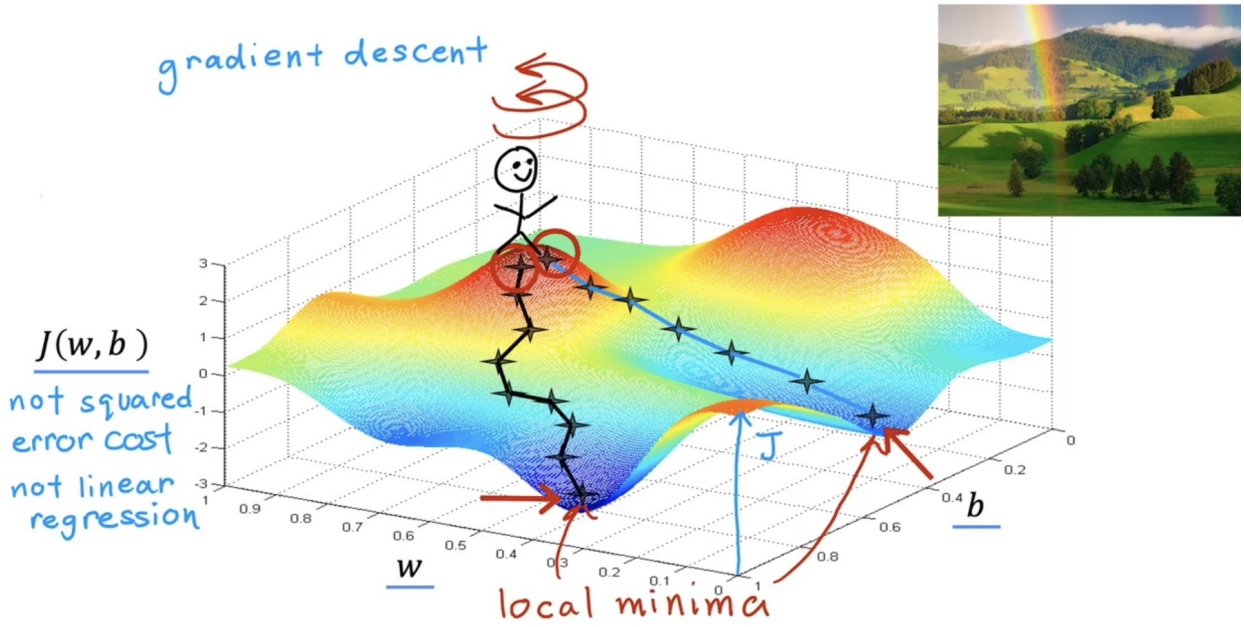




explaination The 3D plot can be transformed into a contour plot.

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

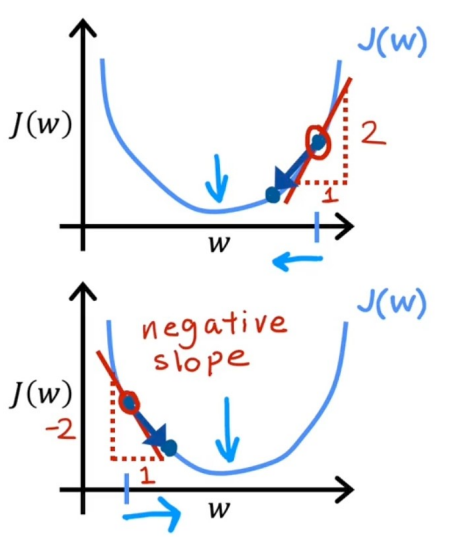
repeat until convergence

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

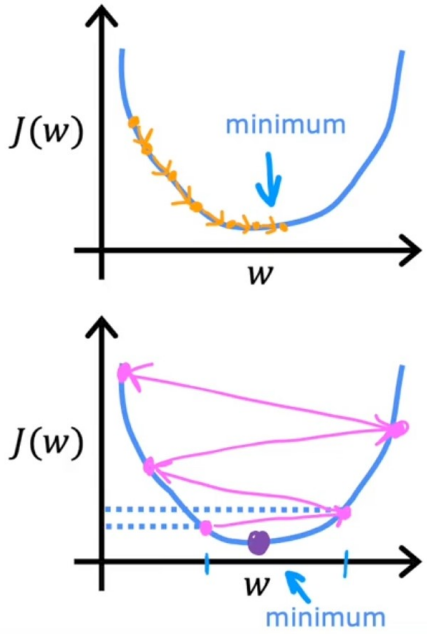
$$b = b - \alpha \cdot \frac{\partial}{\partial b} J(w, b) \tag{3.2}$$

- 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

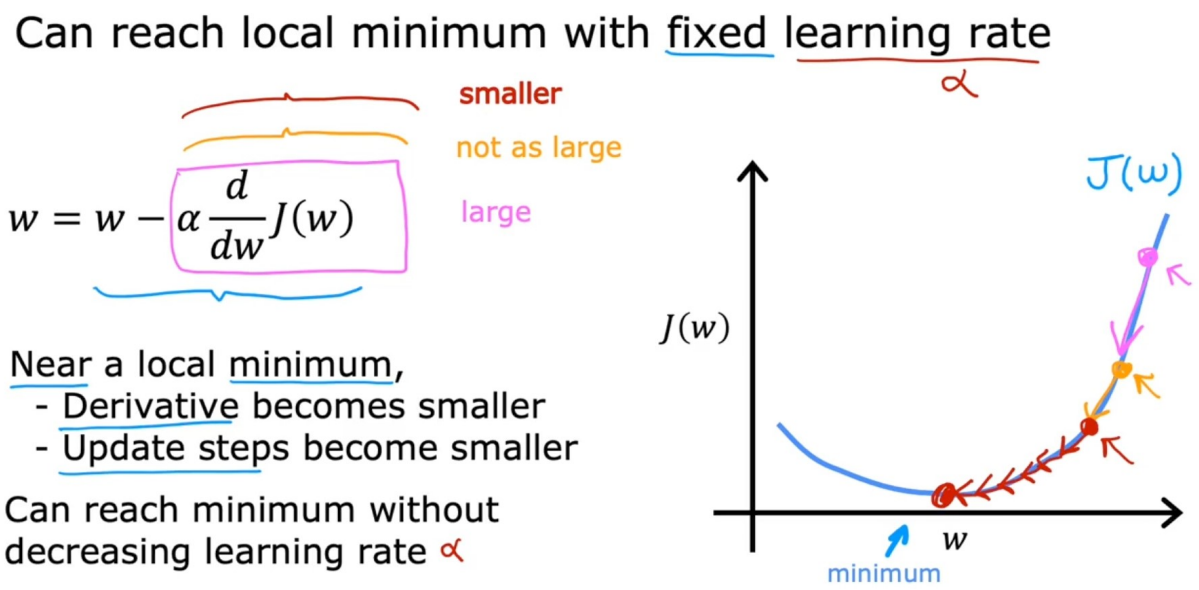


1. If the learning rate is too small, the algorithm will take a long time to converge
2. If the learning rate is too large, the algorithm may overshoot the minimum or even diverge.

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

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

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
```

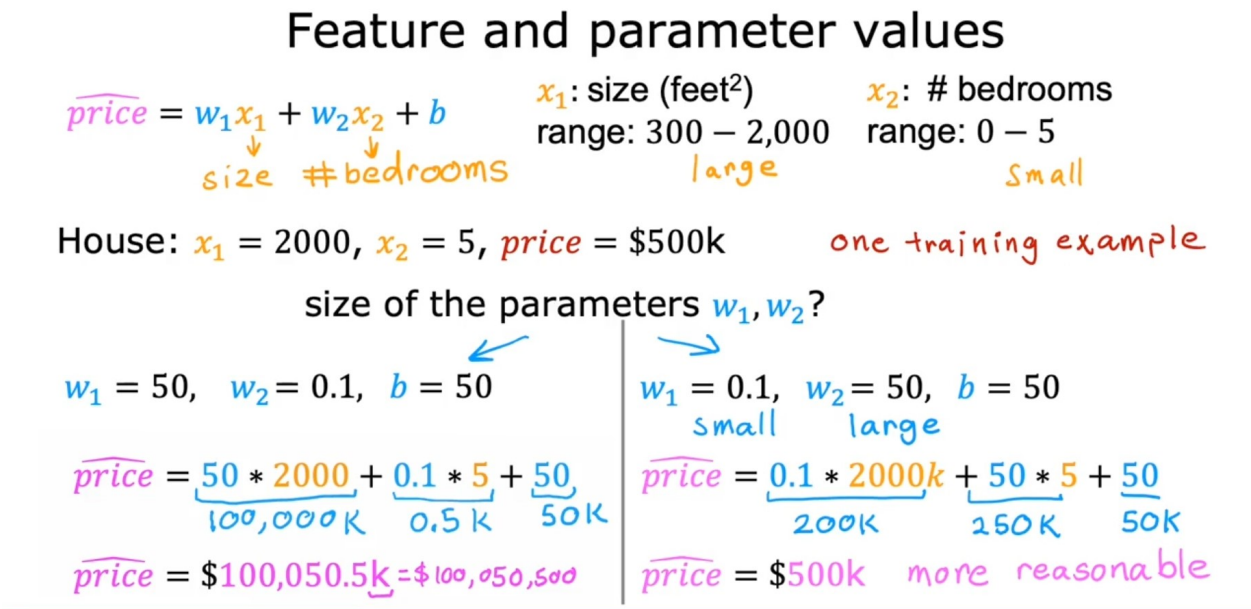
```
      """
    w = deepcopy(w_in)  # avoid modifying global w within function
    b = b_in

    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
```

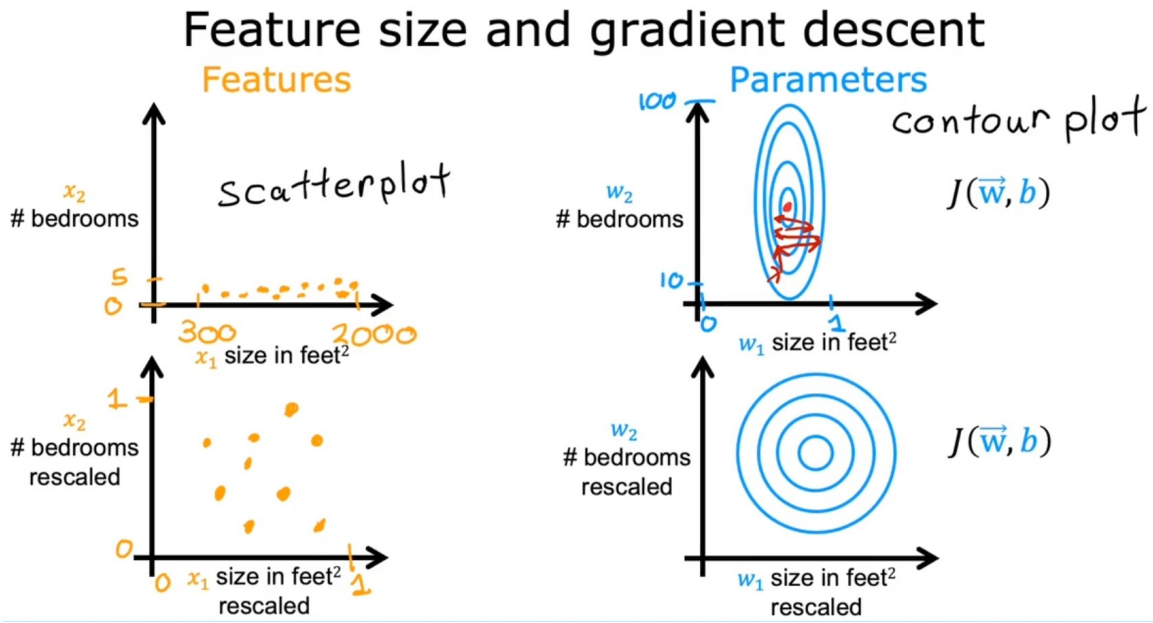
```
    """
    # avoid division by zero
    eps = 1e-8
    for i in range(X.shape[1]):
        X[:, i] = (X[:, i] - np.mean(X[:, i])) / (np.std(X[:, i]) + eps)

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()
```

5

Tips for Gradient Descent

5.1 Feature scaling





Feature scaling

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

$-3 \leq x_j \leq 3$

$-0.3 \leq x_j \leq 0.3$

} acceptable ranges

$0 \leq x_1 \leq 3$	okay, no rescaling
$-2 \leq x_2 \leq 0.5$	okay, no rescaling
$-100 \leq x_3 \leq 100$	too large → rescale
$-0.001 \leq x_4 \leq 0.001$	too small → rescale
$98.6 \leq x_5 \leq 105$	too large → rescale

5.2 Normalization

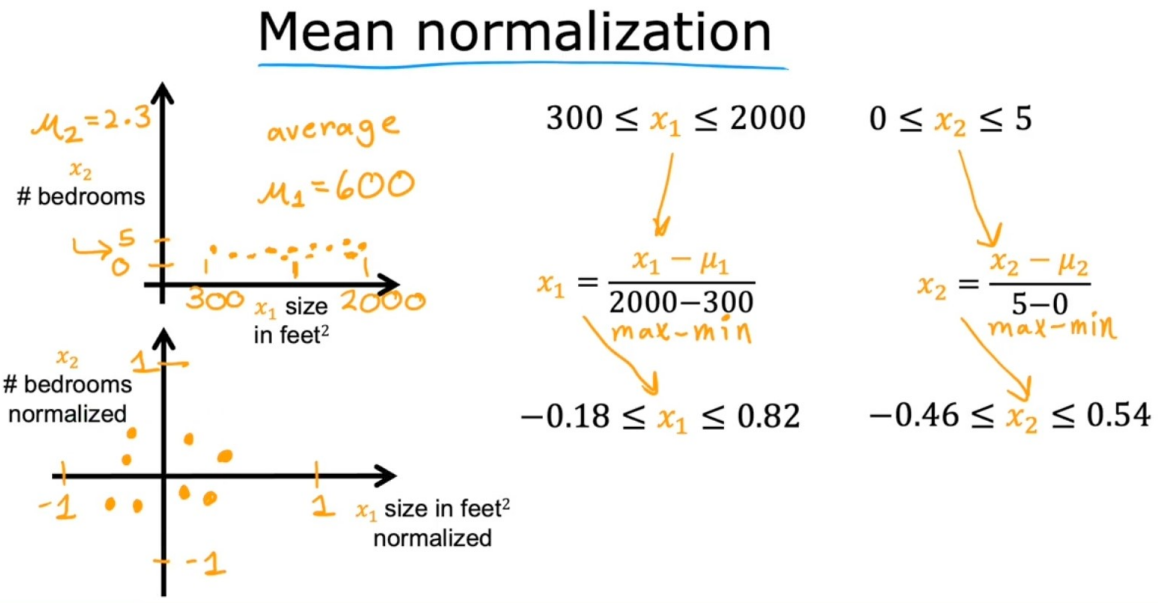
Mean normalization

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

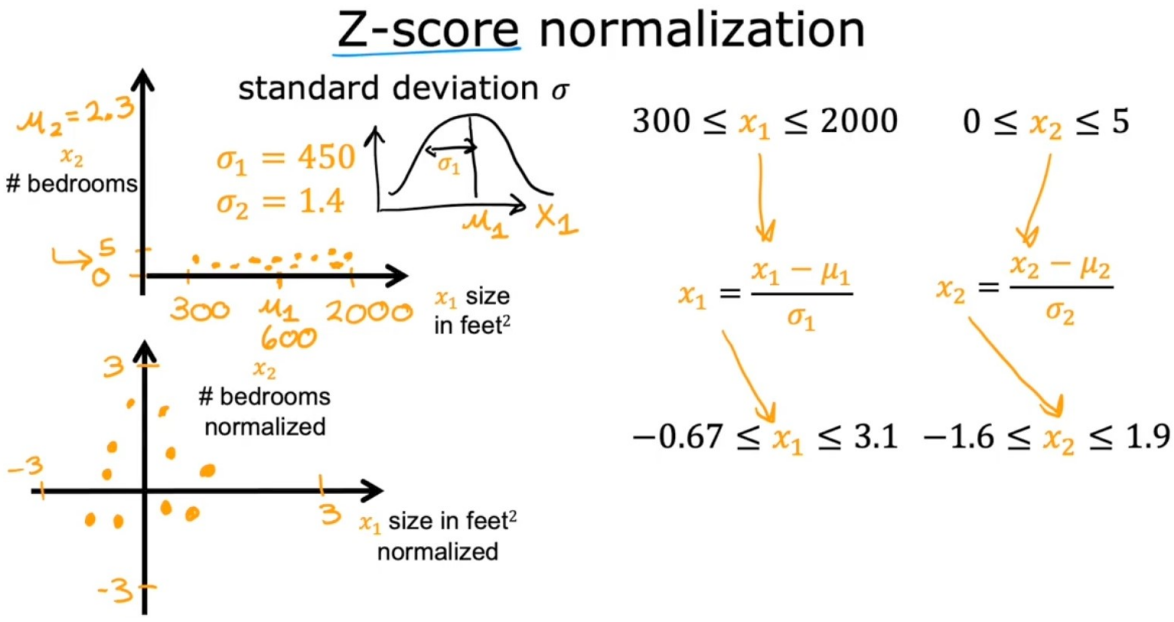
for each feature x_i

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

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

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

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



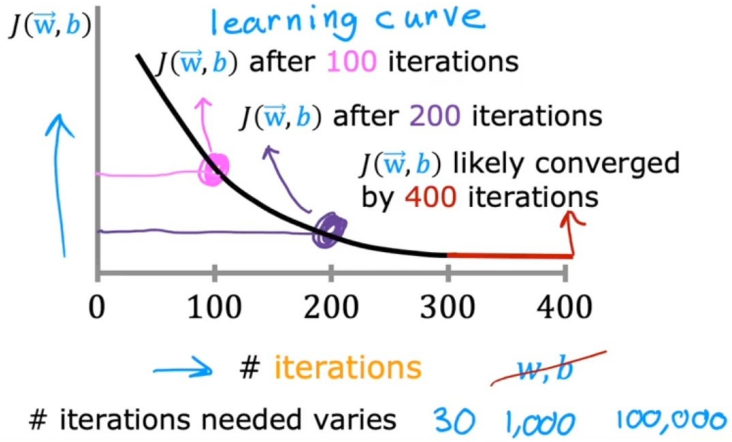
5.3 Check the convergence

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.

Make sure gradient descent is working correctly

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

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



iterations needed varies 30 1,000 100,000

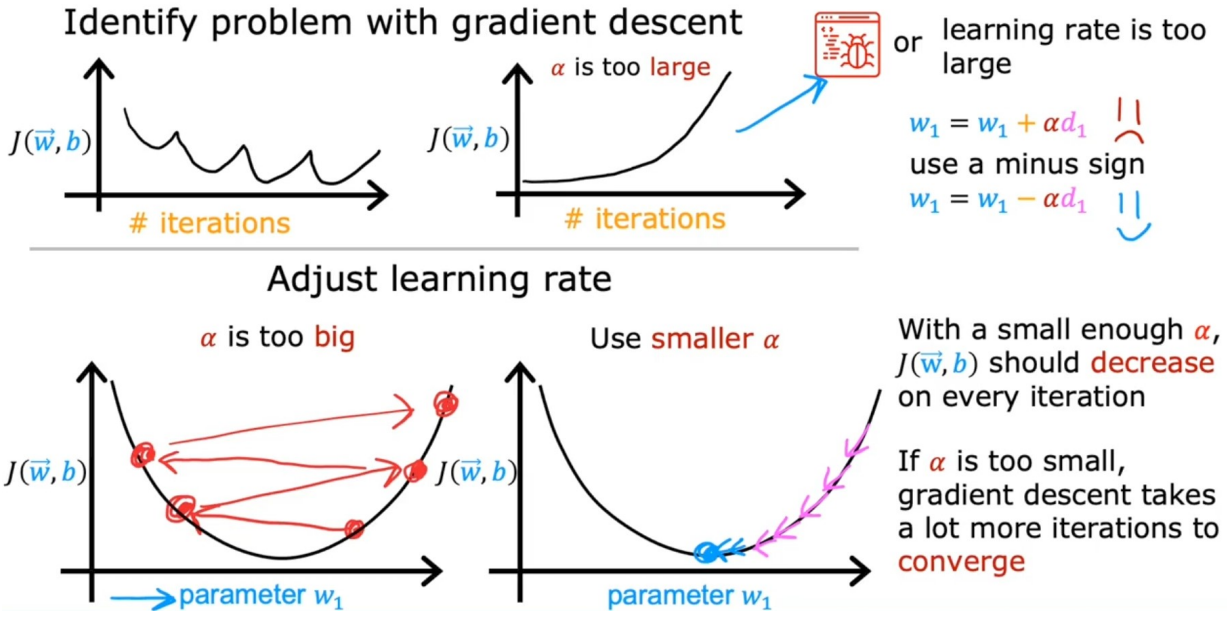
Automatic convergence test

Let ε "epsilon" be 10^{-3}. 0.001

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

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

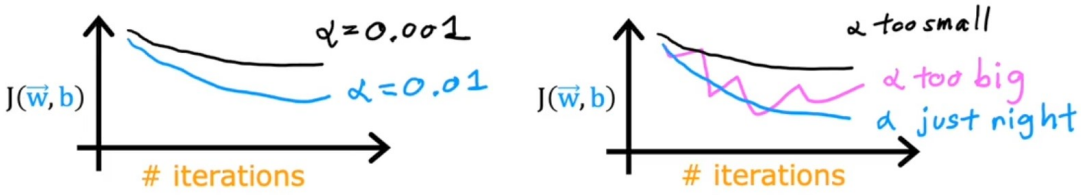
5.4 Choose learning rate



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.

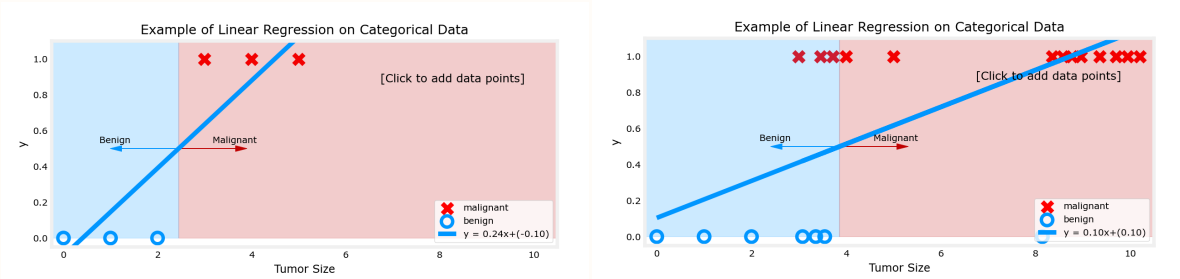
6.1 Linear Regression approach to Classification

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 malignant tumors to be classified as benign. However, the malignant tumors are still malignant, but they are classified as benign because of the shift in the line.

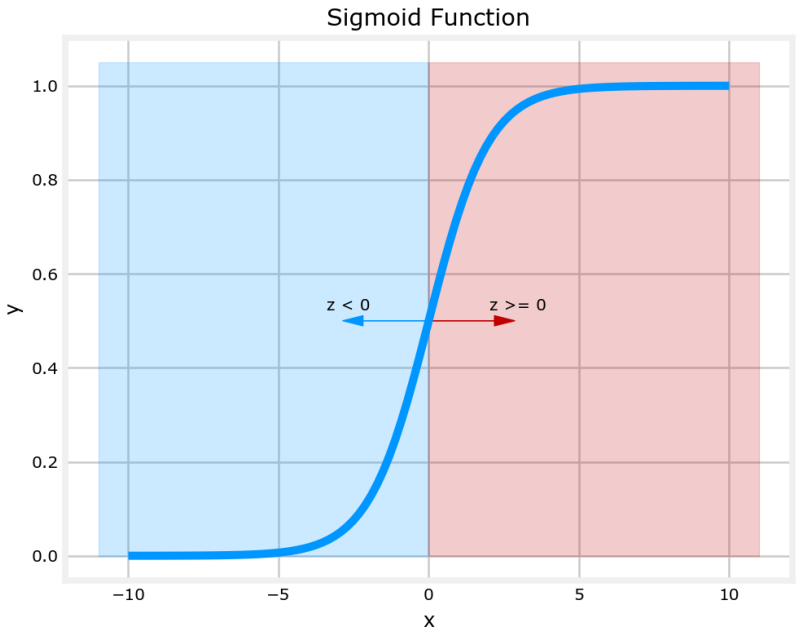
6.2 Logistic Regression

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

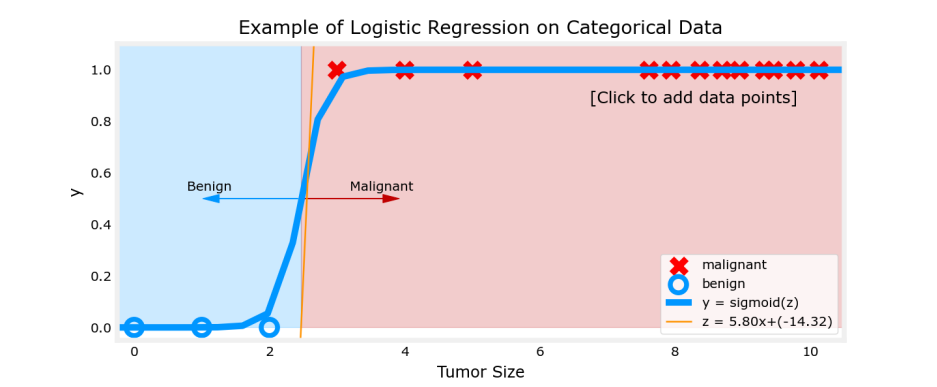
We can transform the output of linear regression using the sigmoid function.

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

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

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

By using the sigmoid function, the output of logistic regression is always between 0 and 1. And it provide a nonlinear fit curve, which solves the problem of linear regression in the previous tumor example.

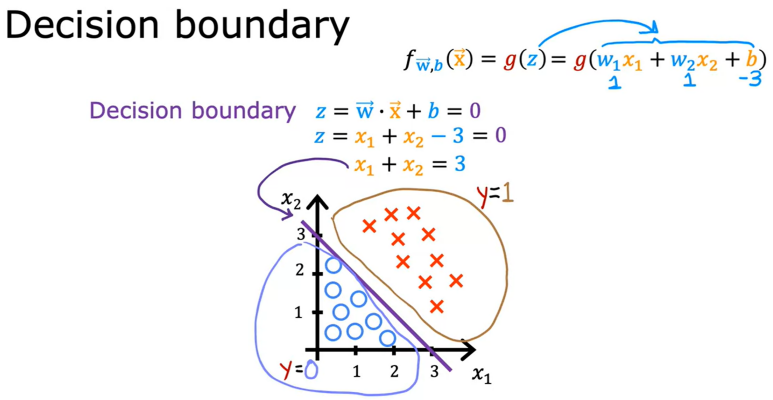


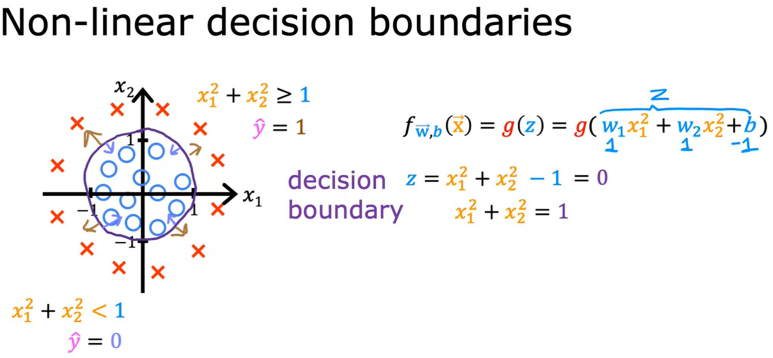
6.3 Decision Boundary

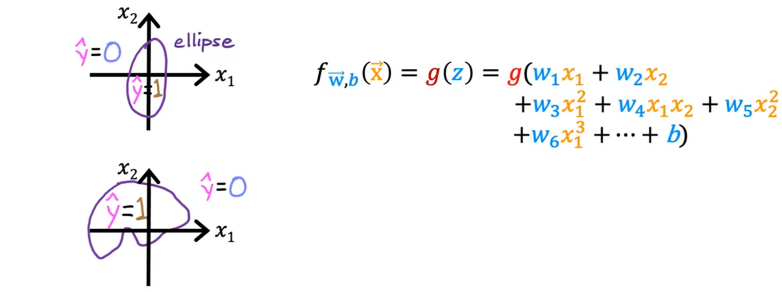
How to classify? when the output of logistic regression is greater than 0.5, we classify it as 1, and when it is less than 0.5, we classify it as 0.

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.







6.4 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 6.4.1 ▶ 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 6.4.2 ▶ 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 (MLE).

In this derivation, we will use the Bernoulli distribution to model the probability of the output.

$$P(Y = 1|X = \mathbf{x}) = f_{\mathbf{w},b}(\mathbf{x}) = \hat{p}$$
$$P(Y = 0|X = \mathbf{x}) = 1 - f_{\mathbf{w},b}(\mathbf{x}) = 1 - \hat{p}$$

Since this PMF of Bernoulli distribution is inderivable, we create a likelihood function: $p^y(1-p)^{1-y}$.

$$P(Y = y|X = \mathbf{x}) = f_{\mathbf{w},b}(\mathbf{x})^y(1 - f_{\mathbf{w},b}(\mathbf{x}))^{1-y} = \hat{p}^y(1-\hat{p})^{1-y}$$

The likelihood function is the product of the probabilities of the observed data points. Assume that the data points are independent and identically distributed (i.i.d):

$$L(\mathbf{w}, b) = \prod_{i=1}^{m} P(Y = y^{(i)}|X = \mathbf{x}^{(i)}) = \prod_{i=1}^{m} \hat{p}^{y^{(i)}}(1-\hat{p})^{1-y^{(i)}}$$

Take the log:

$$\log L(\mathbf{w}, b) = \sum_{i=1}^{m} y^{(i)} \log \hat{p} + (1 - y^{(i)}) \log(1 - \hat{p})$$

This is the log likelihood function. We want to maximize this function, so we take the negative of it as the cost function.

6.5 Gradient Descent

Theorem 6.5.1 ▸ 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:

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

where α is the learning rate. final version:

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

The math derivation is below

key: $\frac{\partial}{\partial z} g(z) = g(z)(1 - g(z))$, $g(z)$ is the sigmoid function.

(actually here z is $z^{(i)}$, but for shorter we write it as z)

let:

$$
\begin{aligned}
z &= \mathbf{w}^T \mathbf{x}^{(i)} + b = w_1 x_1^{(i)} + w_2 x_2^{(i)} + \cdots + w_n x_n^{(i)} \\
f_{\mathbf{w},b}(\mathbf{x}) &= g(z) = \frac{1}{1 + e^{-z}} = p
\end{aligned}
$$

$$
\begin{aligned}
&\frac{\partial}{\partial w_j} J(\mathbf{w}, b) \\
&= \frac{\partial}{\partial w_j} - \frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log p + (1 - y^{(i)}) \log(1 - p) \right] \\
&= \frac{\partial}{\partial w_j} - \frac{1}{m} \sum_{i=1}^{m} \left[\log(1 - p) + y^{(i)} \log \frac{p}{1 - p} \right] \\
&* \; p = \frac{1}{1 + e^{-z}} \\
&= \frac{\partial}{\partial w_j} - \frac{1}{m} \sum_{i=1}^{m} \left[\log \frac{1}{e^z + 1} + y^{(i)} \log e^z \right] \\
&* \text{ chain rule for } z \to w_j: \\
&= -\frac{1}{m} \sum_{i=1}^{m} \left[\frac{\partial \log \frac{1}{e^z + 1}}{\partial z} \frac{\partial z}{\partial w_j} + y^{(i)} \frac{\partial z}{\partial z} \frac{\partial z}{\partial w_j} \right] \\
&= -\frac{1}{m} \sum_{i=1}^{m} \left[(e^z + 1) \frac{-e_z}{(e^z + 1)^2} \frac{\partial z}{\partial w_j} + y^{(i)} \frac{\partial z}{\partial w_j} \right] \\
&= -\frac{1}{m} \sum_{i=1}^{m} \left[\frac{-1}{1 + e^{-z}} + y^{(i)} \right] \frac{\partial z}{\partial w_j} \\
&* \; \frac{\partial z}{\partial w_j} = x_j^{(i)} \\
&= \frac{1}{m} \sum_{i=1}^{m} \left[\frac{1}{1 + e^{-z}} - y^{(i)} \right] x_j^{(i)} \\
&= \frac{1}{m} \sum_{i=1}^{m} \left[f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)} \right] x_j^{(i)}
\end{aligned}
$$

The same goes to $\frac{\partial}{\partial b} J(\mathbf{w}, b)$:

$$
\begin{aligned}
&\frac{\partial}{\partial b} J(\mathbf{w}, b) \\
&= \frac{1}{m} \sum_{i=1}^{m} \left[\frac{1}{1 + e^{-z}} - y^{(i)} \right] \frac{\partial z}{\partial b} \\
&= \frac{1}{m} \sum_{i=1}^{m} \left[f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)} \right]
\end{aligned}
$$

7

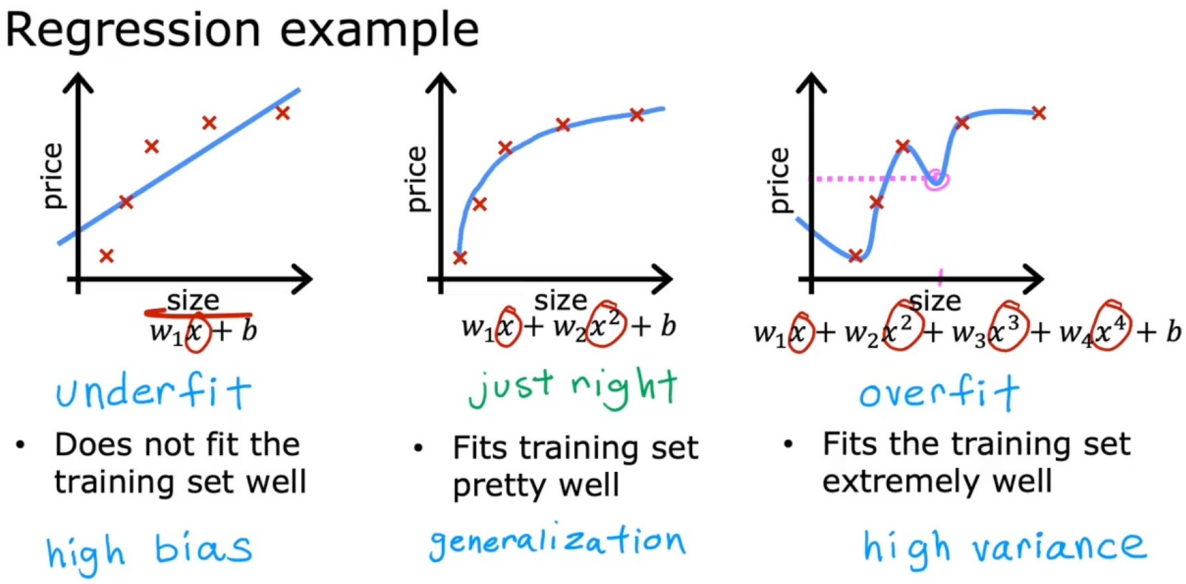
The problem of overfitting

7.1 Overfitting

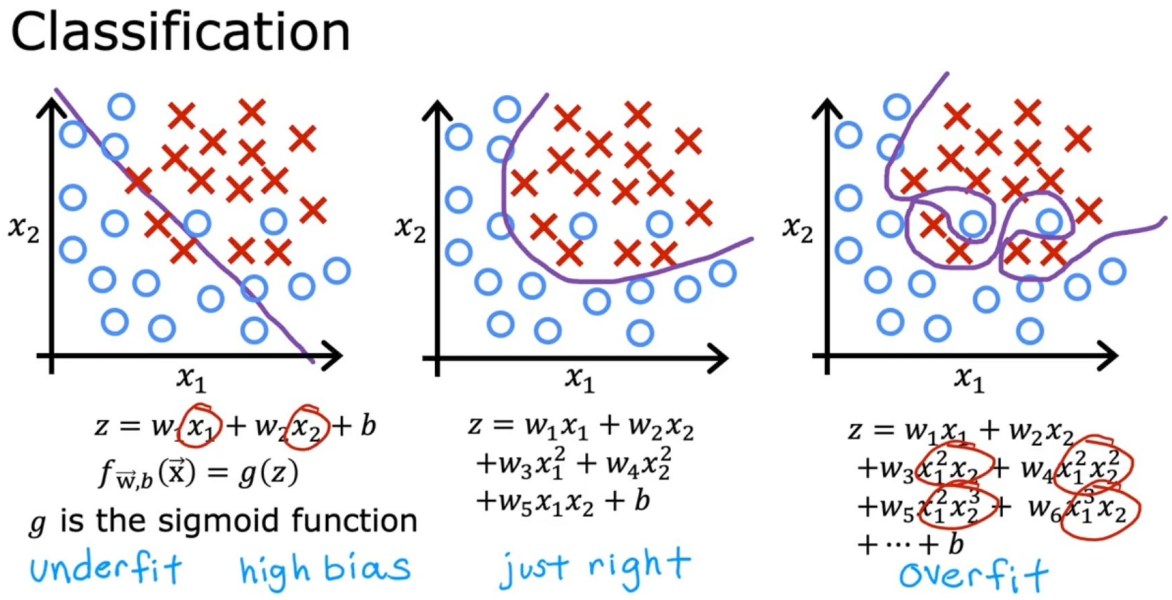
Definition 7.1.1 ▸ overfitting

- **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

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.

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

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**.

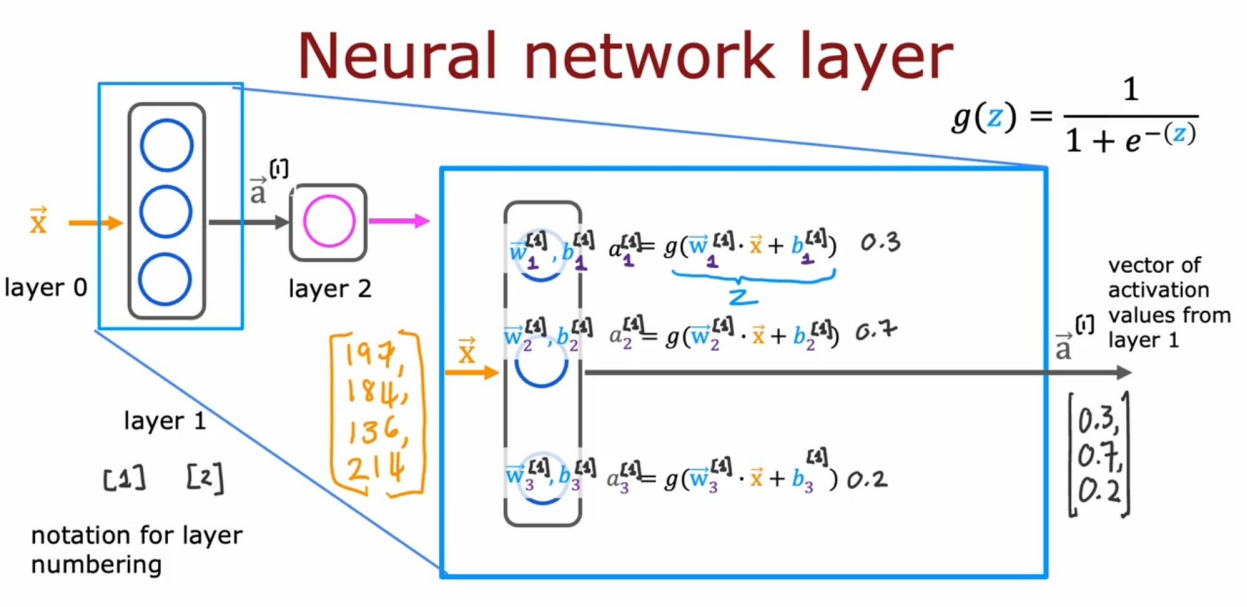
Demand prediction

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

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

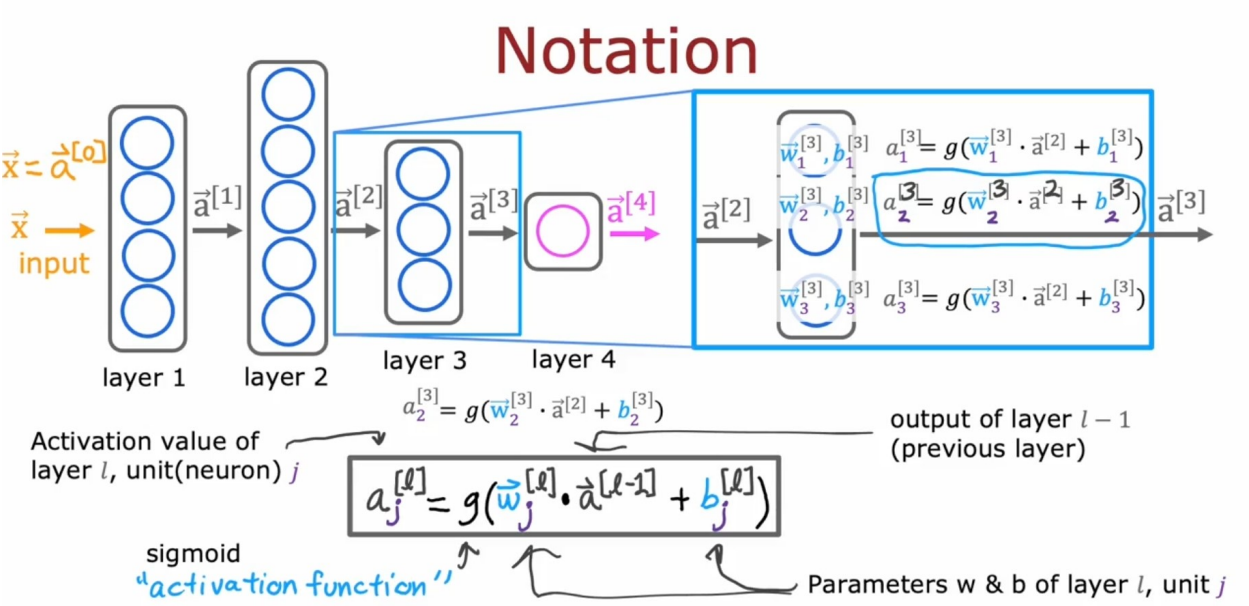
here $\cdot$ is the matrix multiplication, and $g(\mathbf{x})$ means apply g to each element in $\mathbf{x}$.

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".

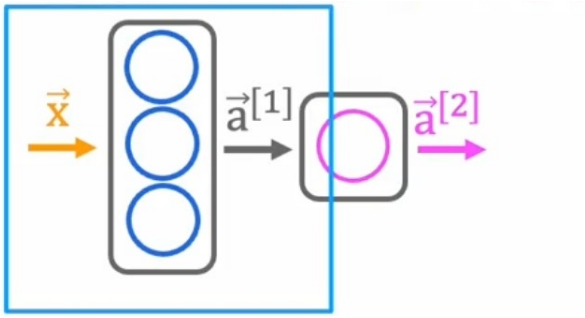
There are many things neural networks can do, such as image recognition, speech recognition, natural language processing, and many other tasks.



8.3 Implementation in TensorFlow

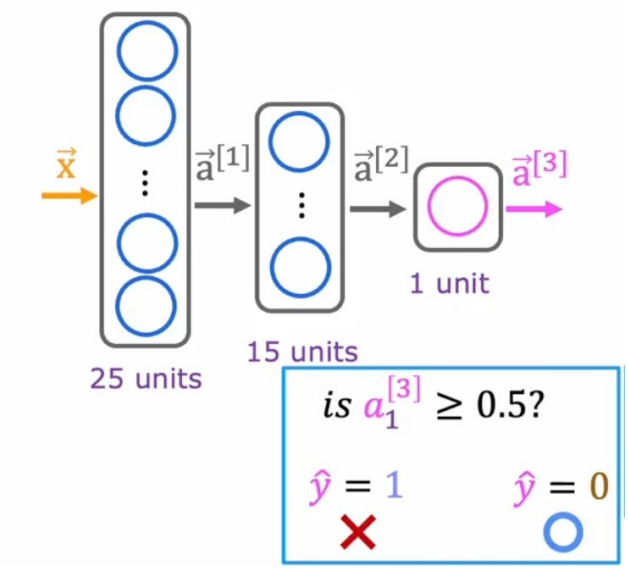
Inference

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:

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

recall: this loss function is derived from the maximum likelihood estimation of the logistic regression model. It is also the loss function of neural networks with sigmoid activation function in the output layer. You can think the output $f_{\mathbf{w},b}(\mathbf{x})$ as $\hat{p}$, and to get $\hat{p}$, there are different approaches, but the likelihood estimation is still the same one. The differences are neural networks have more parameters to compute derivatives and update the parameters.

```
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

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

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 SparseCategoricalCrossentropy
model.compile(loss=SparseCategoricalCrossentropy())
```

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 changing the output layer activation to linear 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)`.

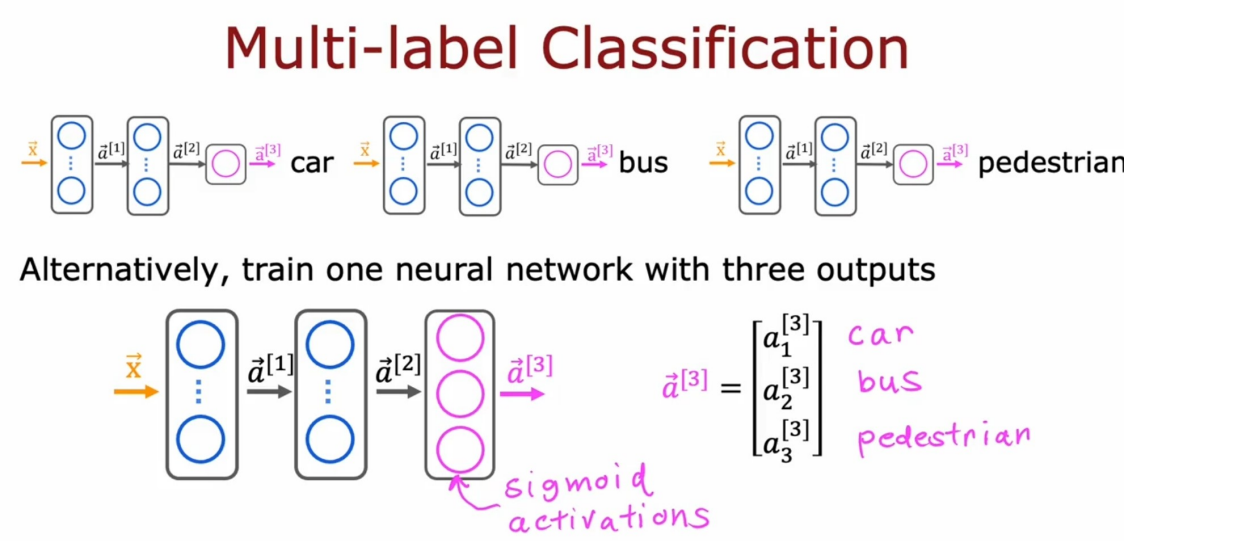
```
model = Sequential([
    Dense(units=25, activation='relu'),
    Dense(units=15, activation='relu'),
    Dense(units=1, activation='linear')
])
model.compile(..., loss=BinaryCrossentropy(from_logits=True))
model.fit(X, Y, epochs=100)

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.



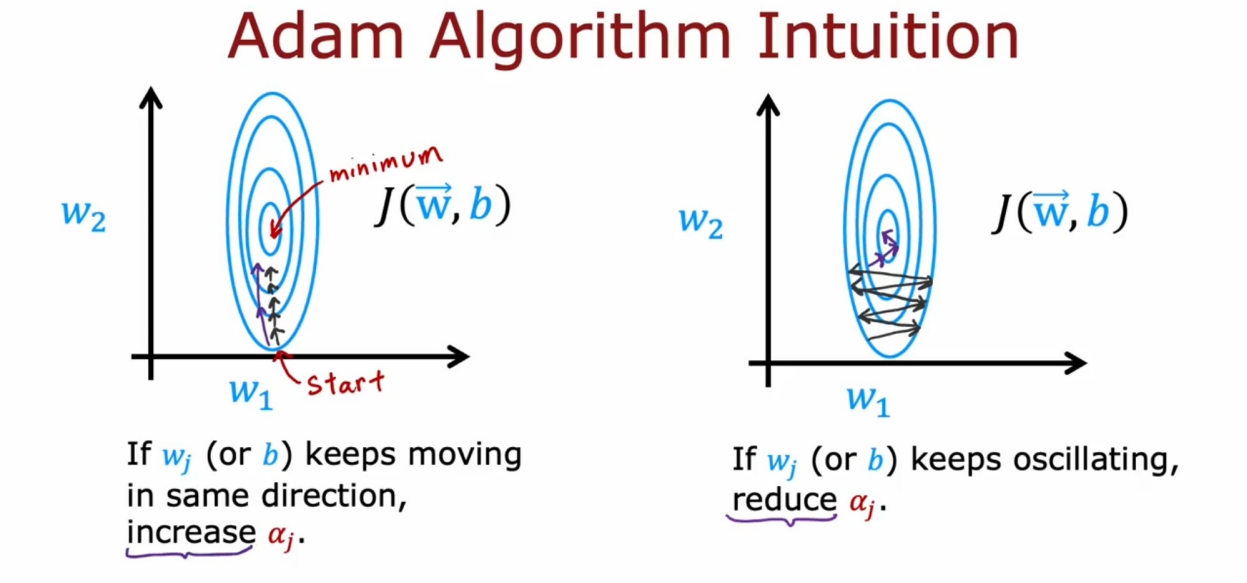
In this example, each image can be assigned to multiple labels. For example, an image can be assigned to both “include car”, “include bus”, and “include pedestrian”. We can build

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.



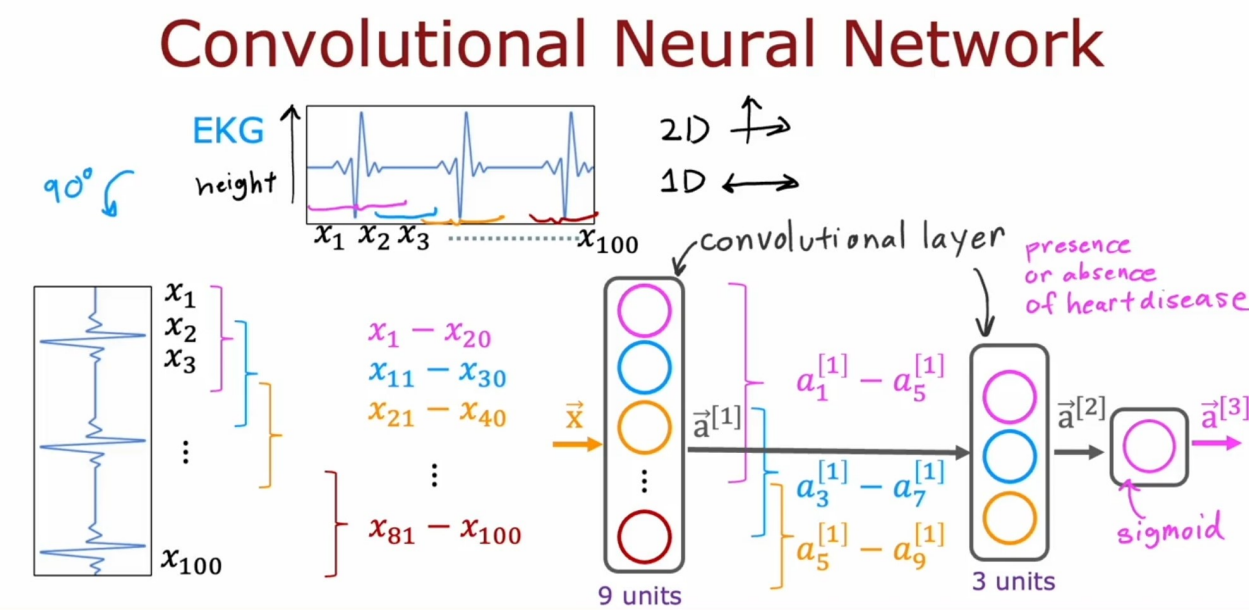
By changing the learning rate, we can achieve better performance.

```
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

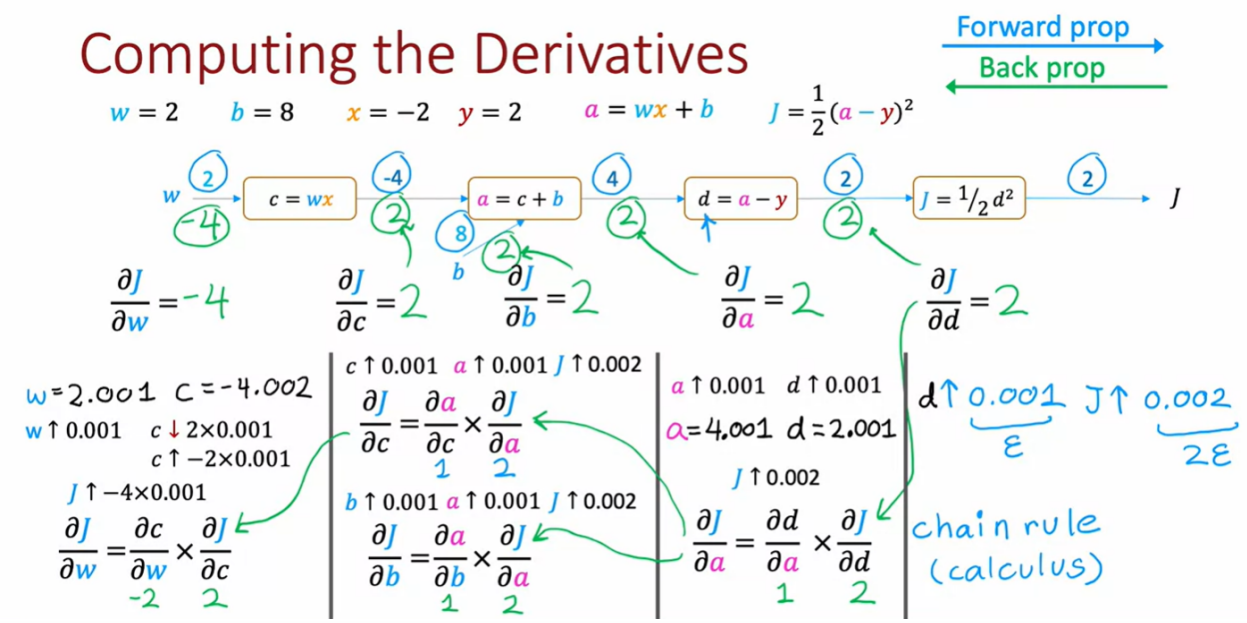
Dense layer: Each neuron output is a function of all the inputs.

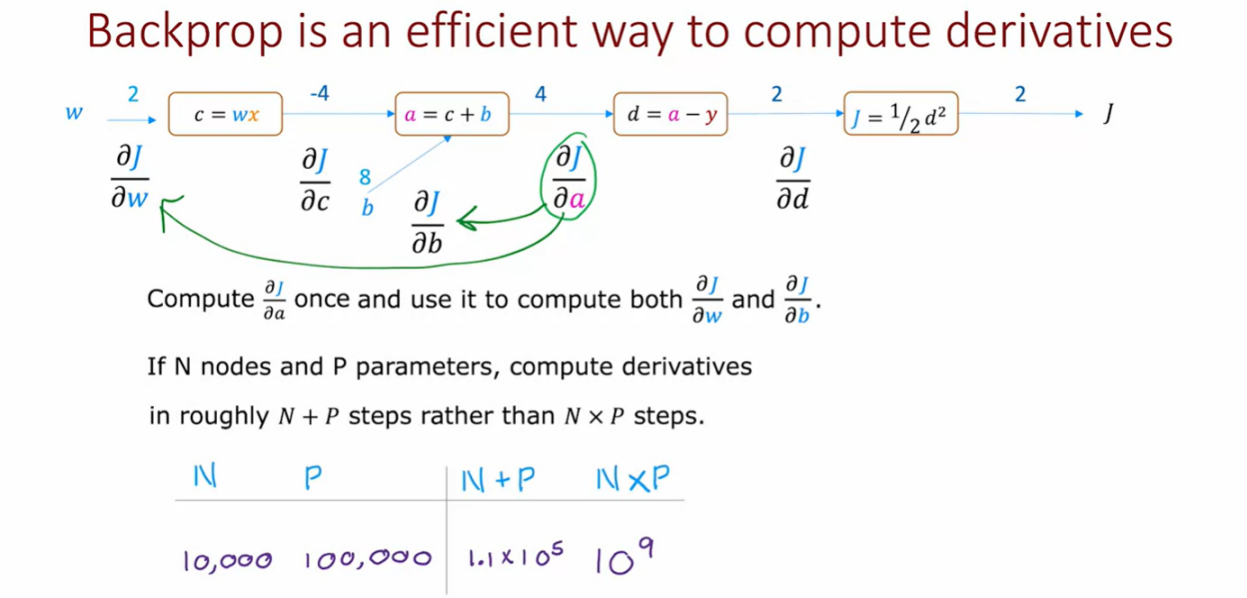
Convolutional layer: Each neuron output is a function of a subset of the inputs. Why? 1. Faster computation. 2. Need less training data (less prone to overfitting).



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.

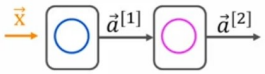
Back propagation is an efficient way to compute the gradients of the loss function with respect to the parameters of the neural network. When there is N nodes and P parameters in the network, the back propagation algorithm only need to compute $O(N+P)$ operations instead of $O(NP)$ operations. The N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

$x = 1 \quad y = 5$

$w^{[1]} = 2, b^{[1]} = 0$ ReLU activation

$w^{[2]} = 3, b^{[2]} = 1$ $g(z) = \max(0, z)$



$a^{[1]} = g(w^{[1]} x + b^{[1]}) = w^{[1]} x + b^{[1]} = 2 \times 1 + 0 = 2$ (2.001, 2.001)

$a^{[2]} = g(w^{[2]} a^{[1]} + b^{[2]}) = w^{[2]} a^{[1]} + b^{[2]} = 3 \times 2 + 1 = 7$ (2.001, 7.003)

$J(w,b) = \frac{1}{2}(a^{[2]} - y)^2 = \frac{1}{2}(7-5)^2 = 2$ (7.003, 2.006005, $(2.003)^2/2$)

$w^{[1]} \uparrow 0.001 \quad J \uparrow 6 \times 0.001 \quad \frac{\partial J}{\partial w^{[1]}} = 6$

$\frac{\partial J}{\partial w^{[1]}} \quad \frac{\partial J}{\partial b^{[1]}} \quad \frac{\partial J}{\partial w^{[2]}} \quad \frac{\partial J}{\partial b^{[2]}}$

N nodes → P parameters $w_1, b_1, w_2, b_2 \ldots$

inefficient way: N × P

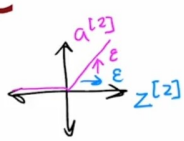
efficient way (backprop): N + P

Neural Network Example

$x = 1 \quad y = 5$

$w^{[1]} = 2, b^{[1]} = 0$ ReLU activation

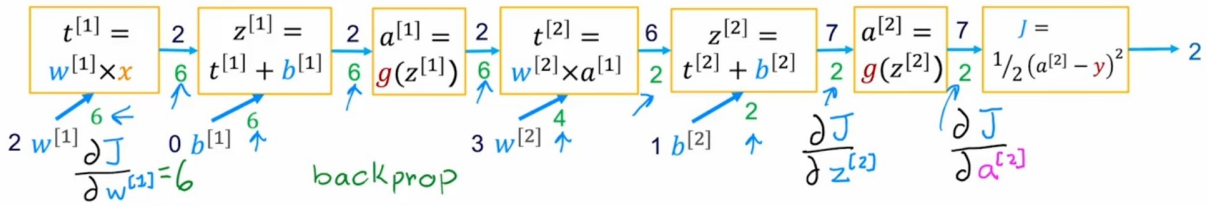
$w^{[2]} = 3, b^{[2]} = 1$ $g(z) = \max(0, z)$



$a^{[1]} = g(w^{[1]} x + b^{[1]}) = \overbrace{w^{[1]} x + b^{[1]}}^{z^{[1]}} = 2 \times 1 + 0 = 2$

$a^{[2]} = g(w^{[2]} a^{[1]} + b^{[2]}) = \overbrace{w^{[2]} a^{[1]} + b^{[2]}}^{z^{[2]}} = 3 \times 2 + 1 = 7$

$J(w,b) = \frac{1}{2}(a^{[2]} - y)^2 = \frac{1}{2}(7-5)^2 = 2$



if $w^{[1]} \uparrow \varepsilon$, then $J \uparrow 6 \times \varepsilon$

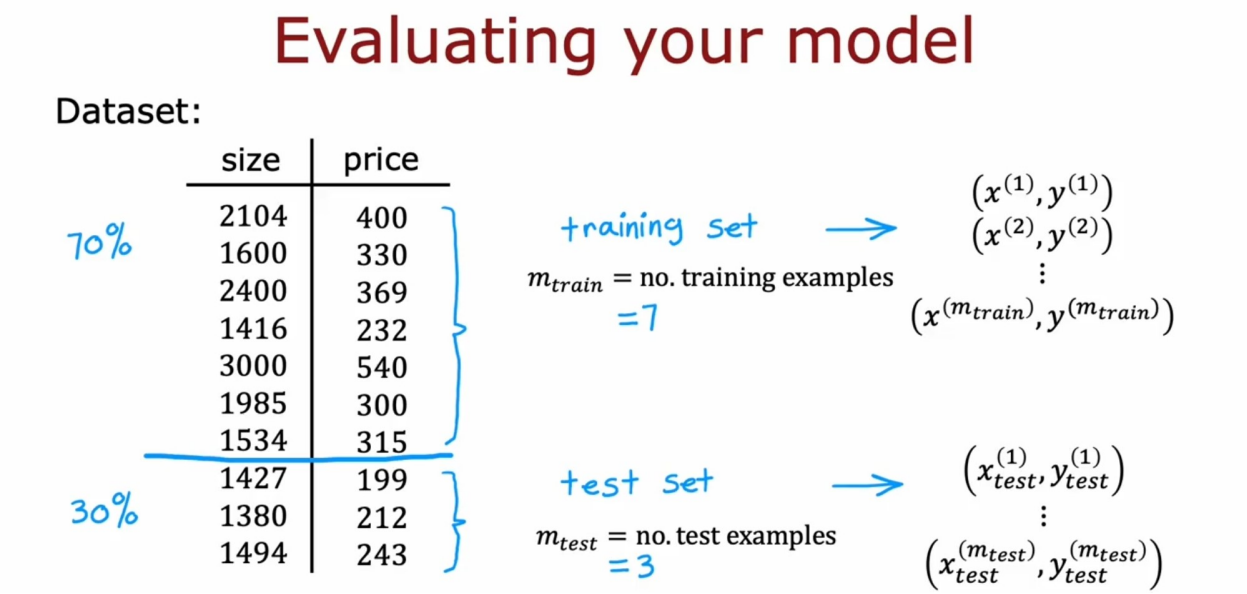
10

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. $J_{\text{test}}(w^{\langle 10 \rangle}, b^{\langle 10 \rangle})$

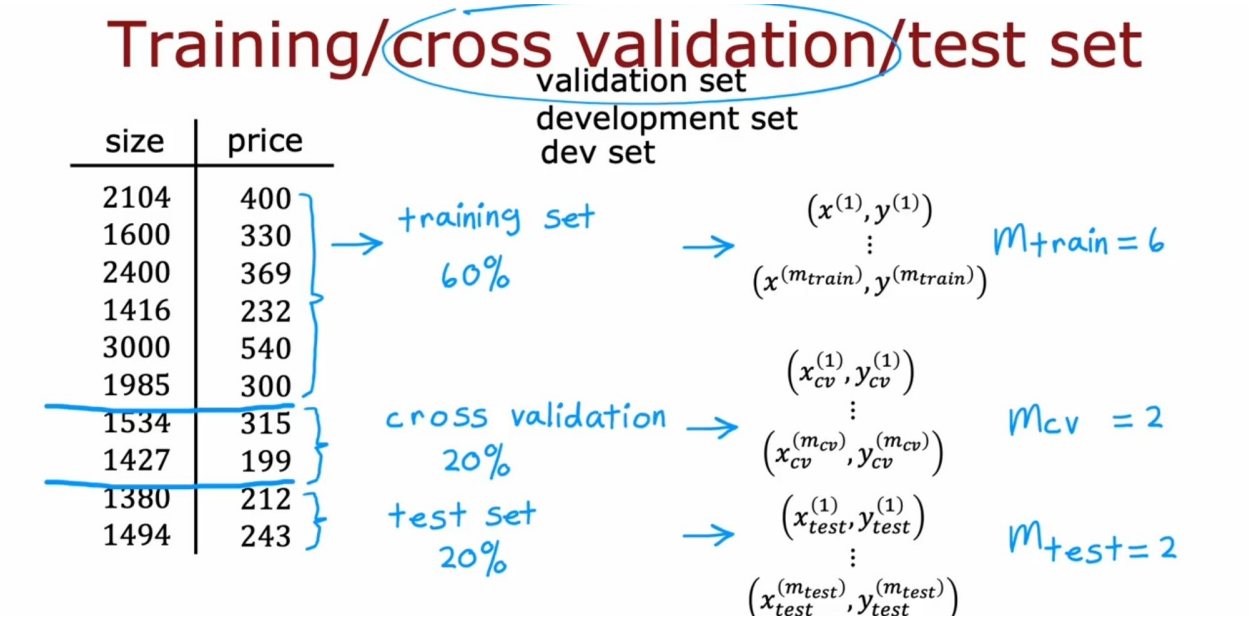
Assume that the $J_{\text{test}}(w^{\langle 5 \rangle}, b^{\langle 5 \rangle})$ is the smallest, then we choose the model with $d = 5$.

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

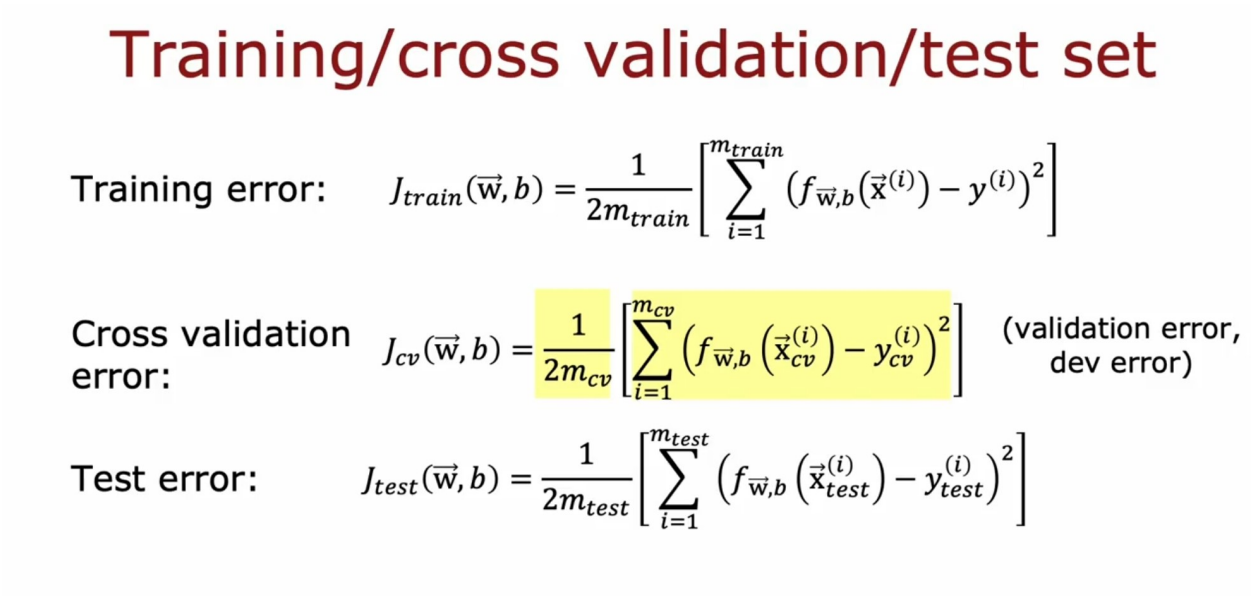
Split the data into three sets: a training set, a cross-validation set, and a testing set.



Compute the error:

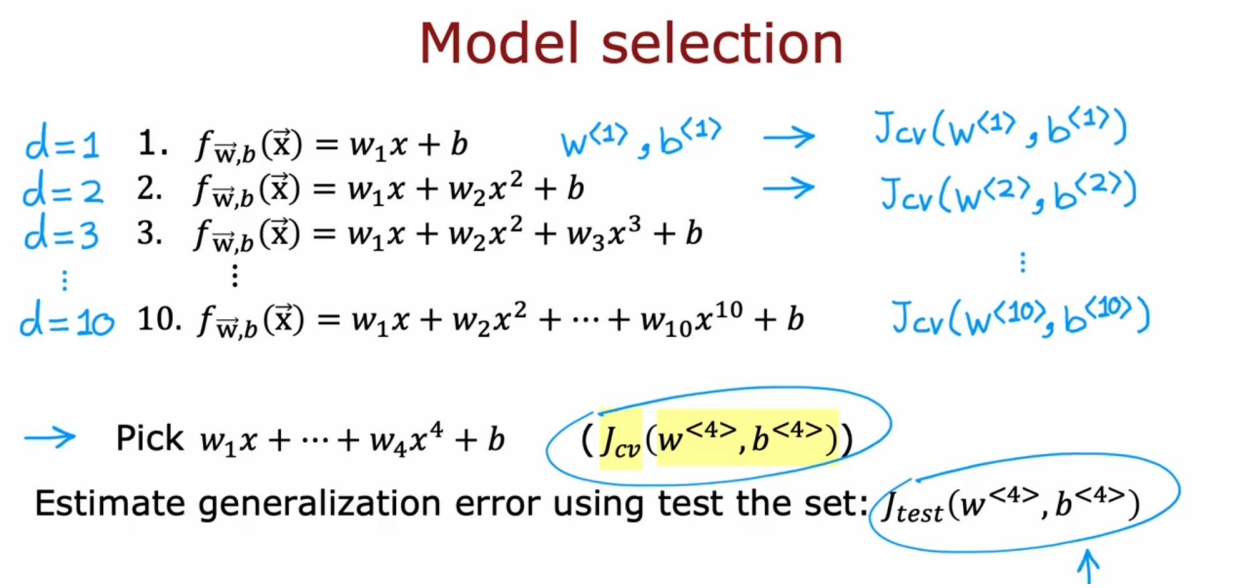
The same method as before, but now we will compute the cross-validation error:

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

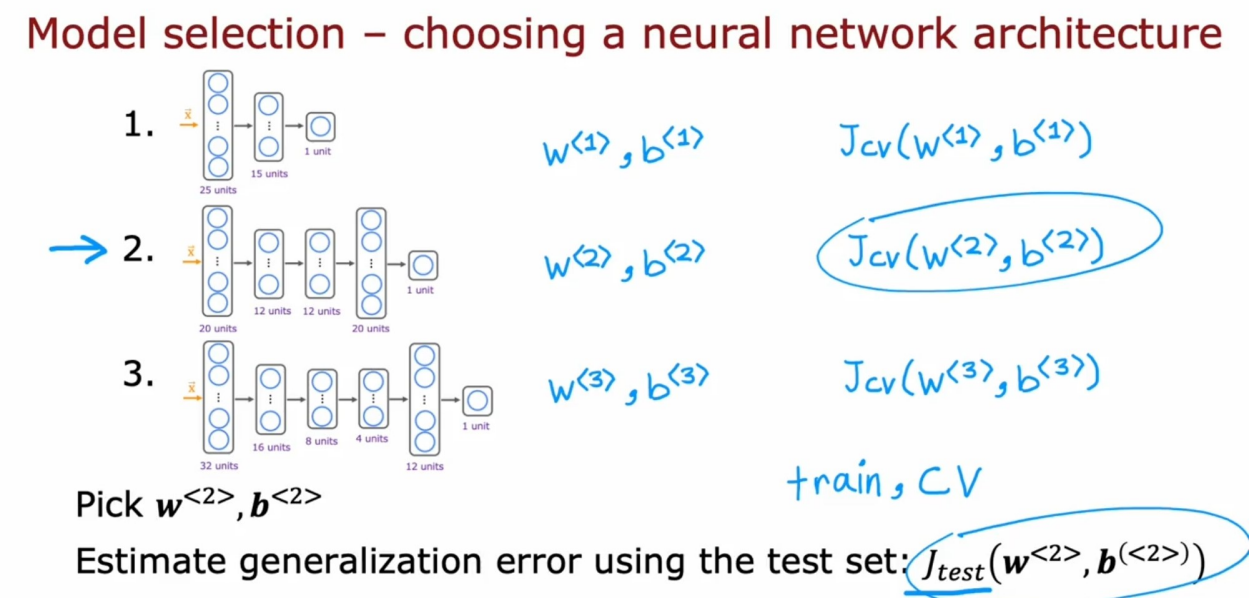


Model selection:

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.



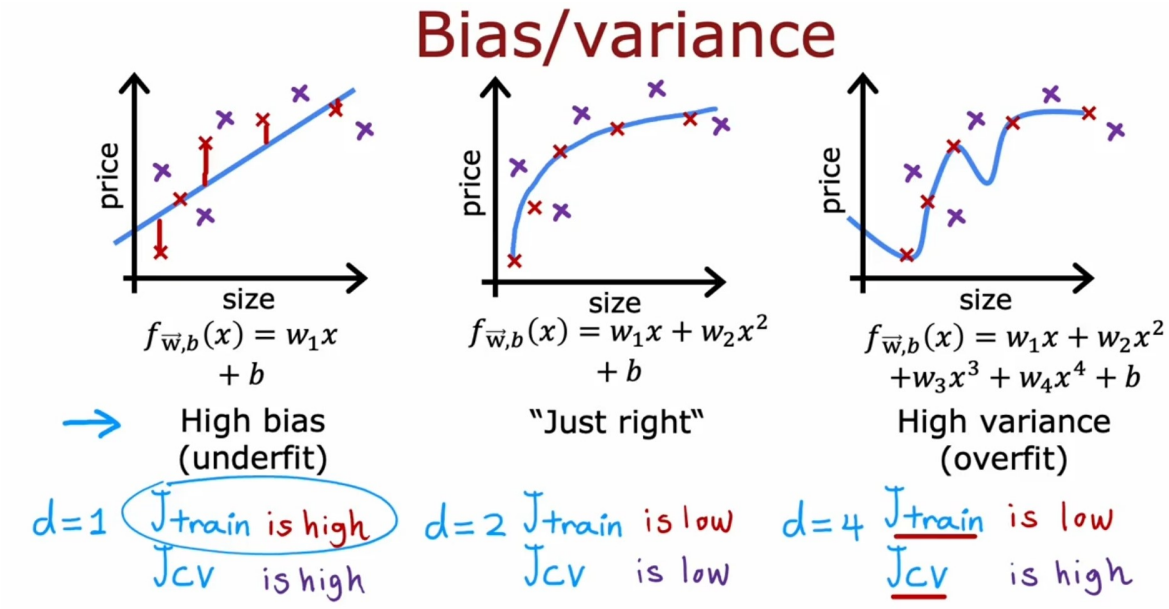
10.2 Bias and variance

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):

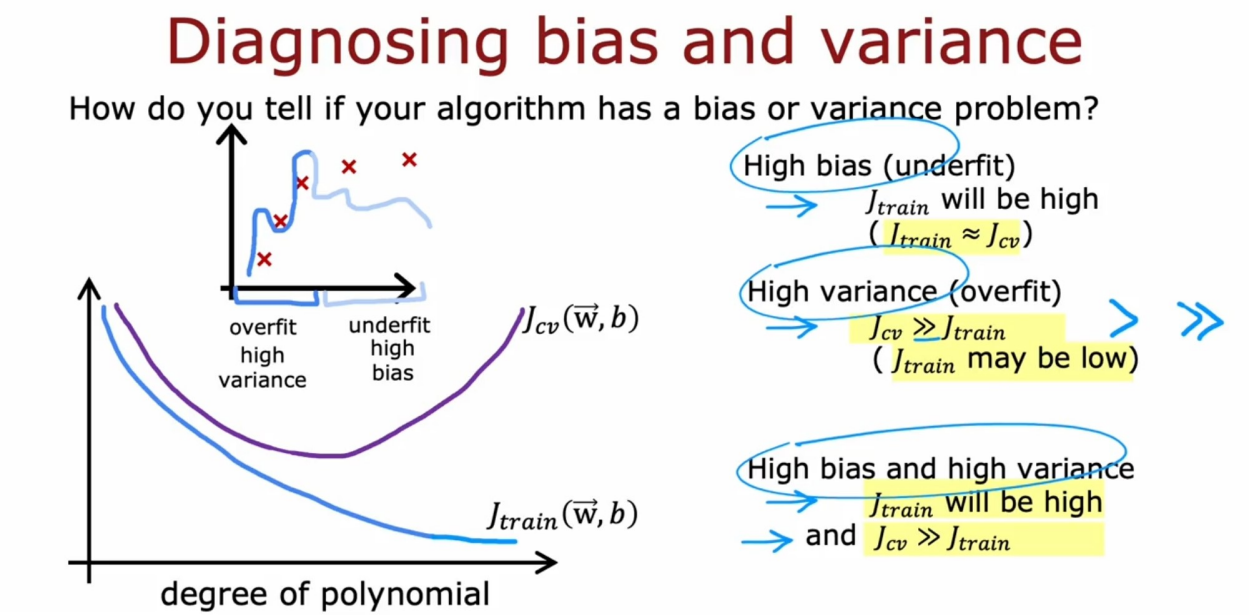
  J_{train} will be high and $J_{\text{cv}} \approx J_{\text{train}}$.

- High Variance (overfit):

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

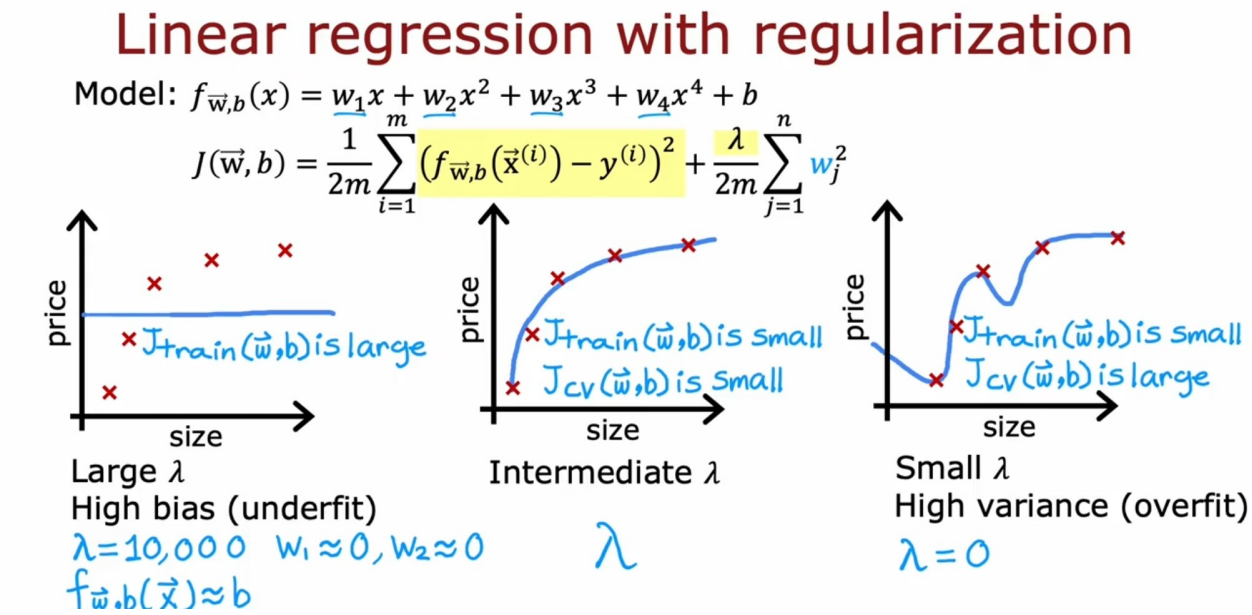
- High Bias and High Variance:

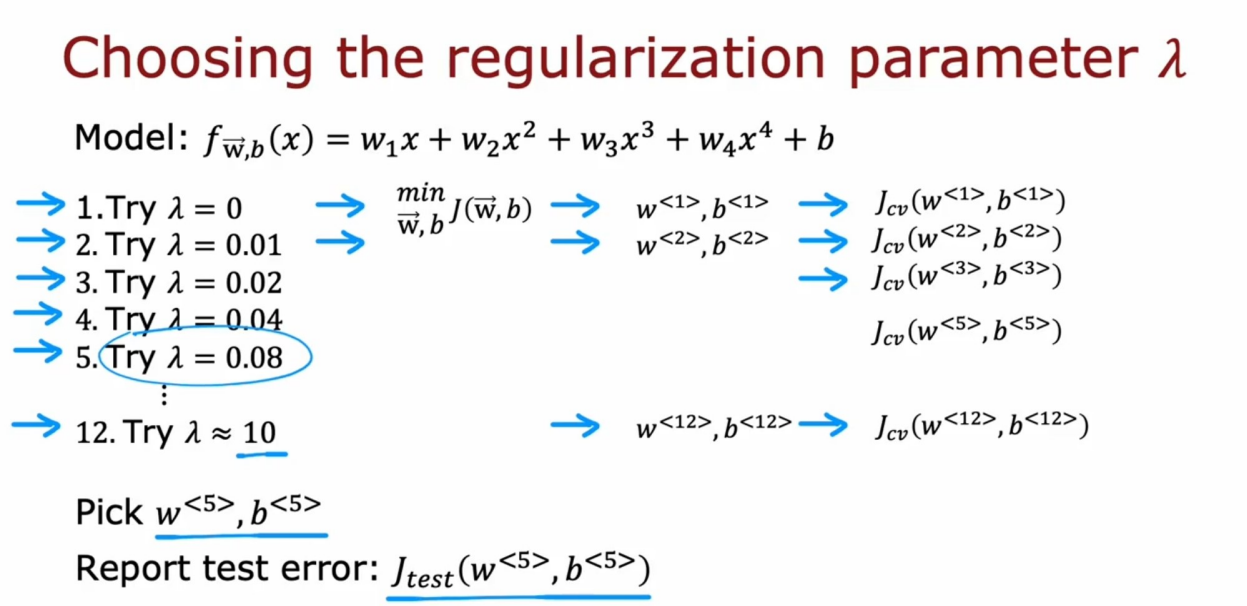
  $J_{\text{cv}} \gg J_{\text{train}}$ and J_{train} is high.



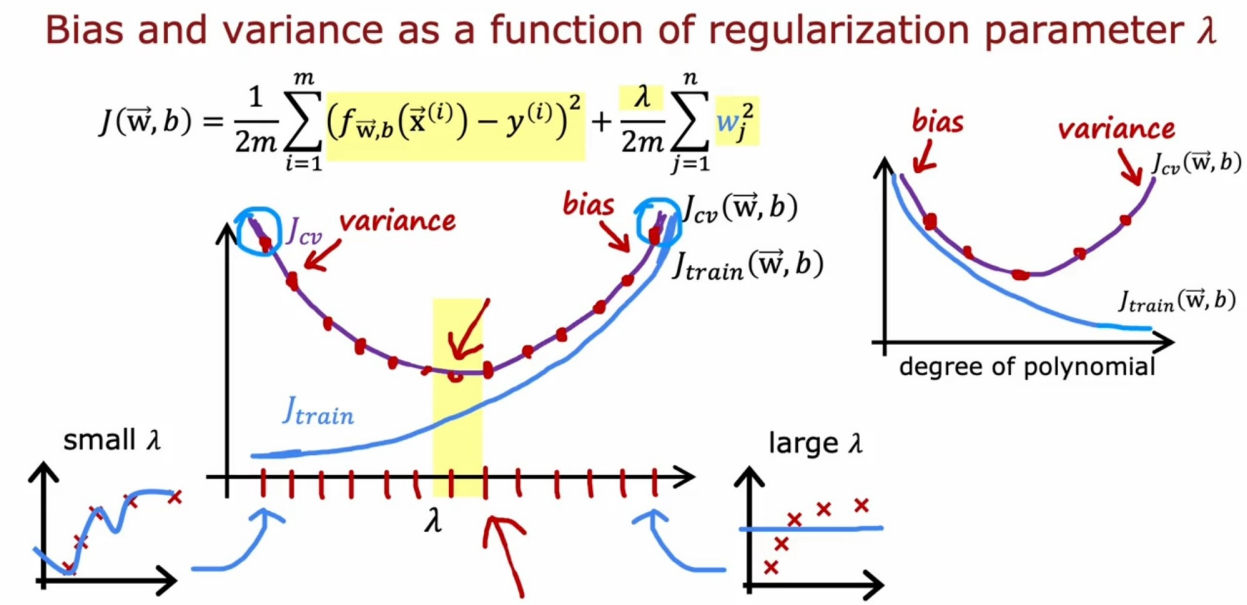
Regularization

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

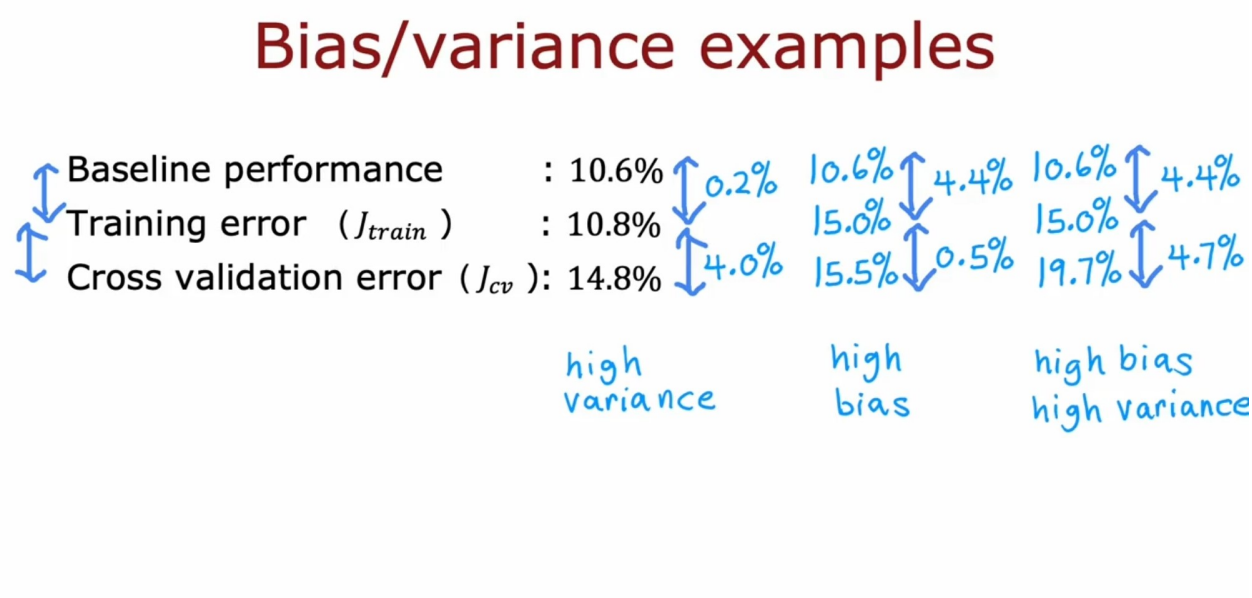
Establishing a baseline level of performance is important. It can be used to compare the performance of your model.

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

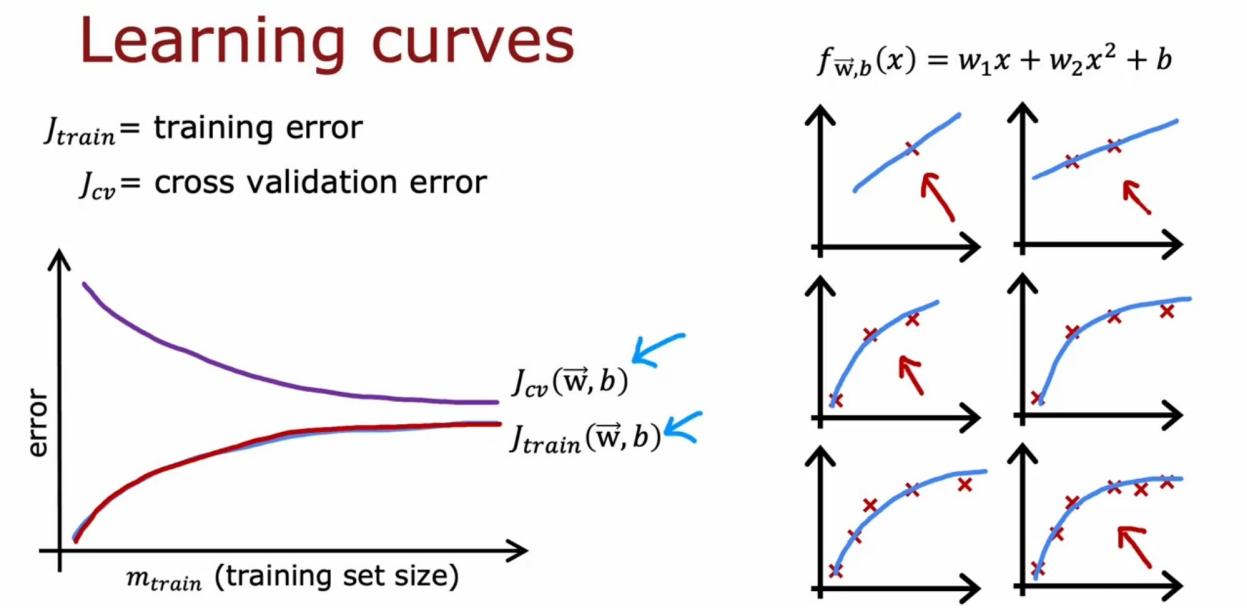
The baseline performance can be used to diagnose bias and variance.

- 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.



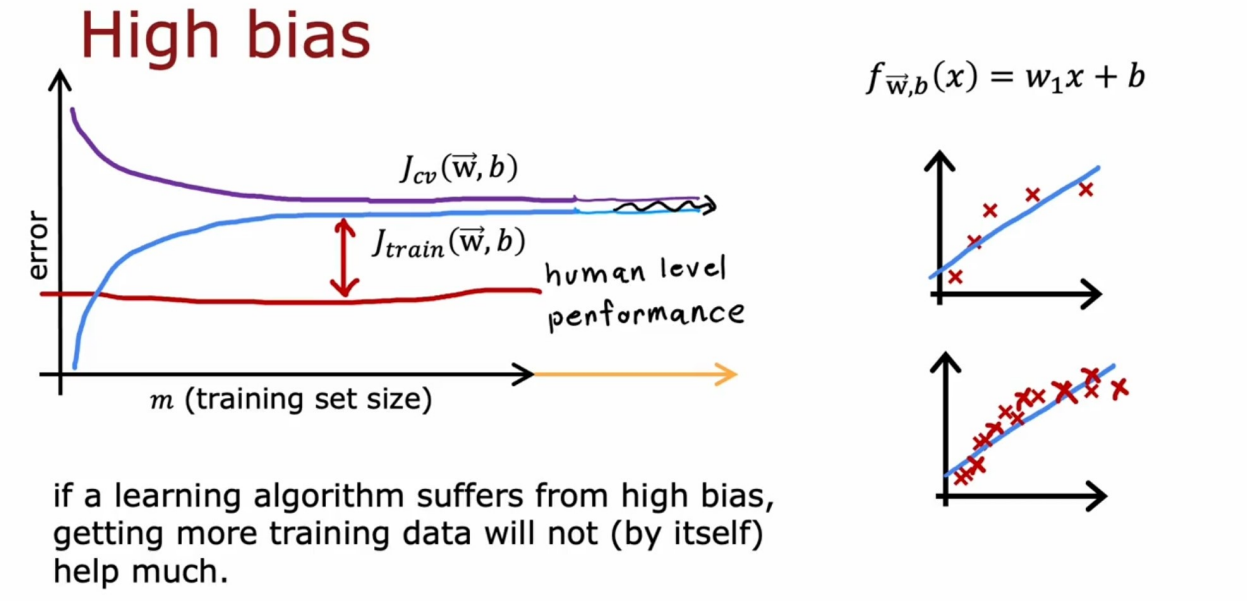
10.4 Learning curves

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



High bias

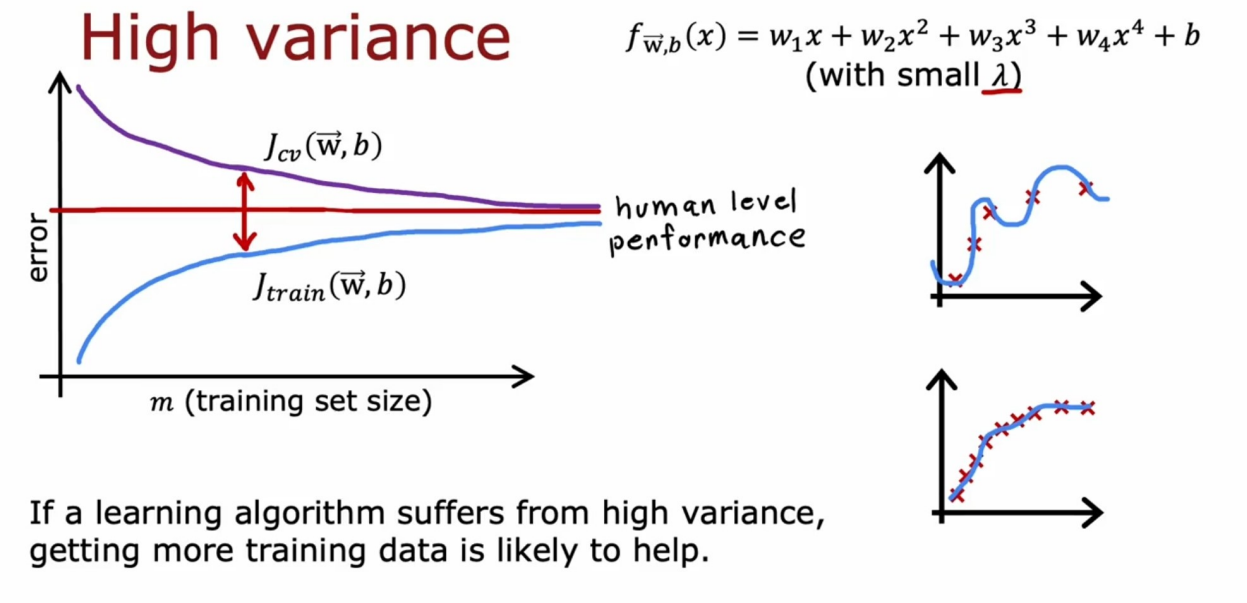
As the training data increases, J_cv and J_train will become flat, but still have a gap beween the baseline performance.



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

High variance

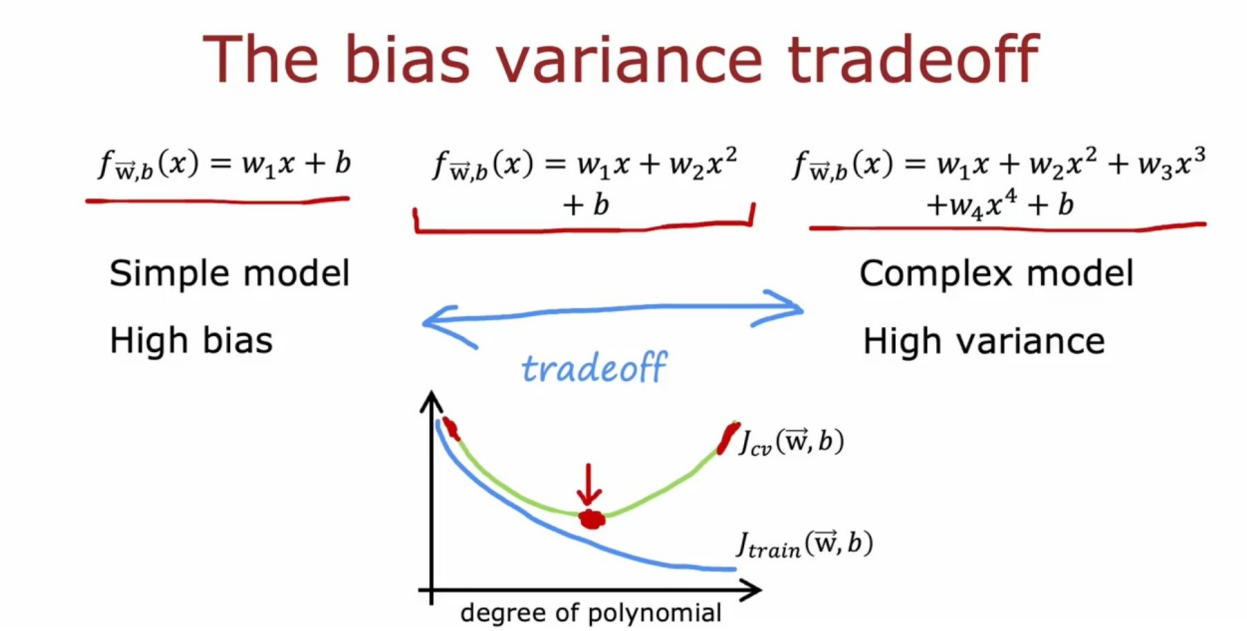
As the traing data increases, J_cv and J_train will become closer to the baseline performance.



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

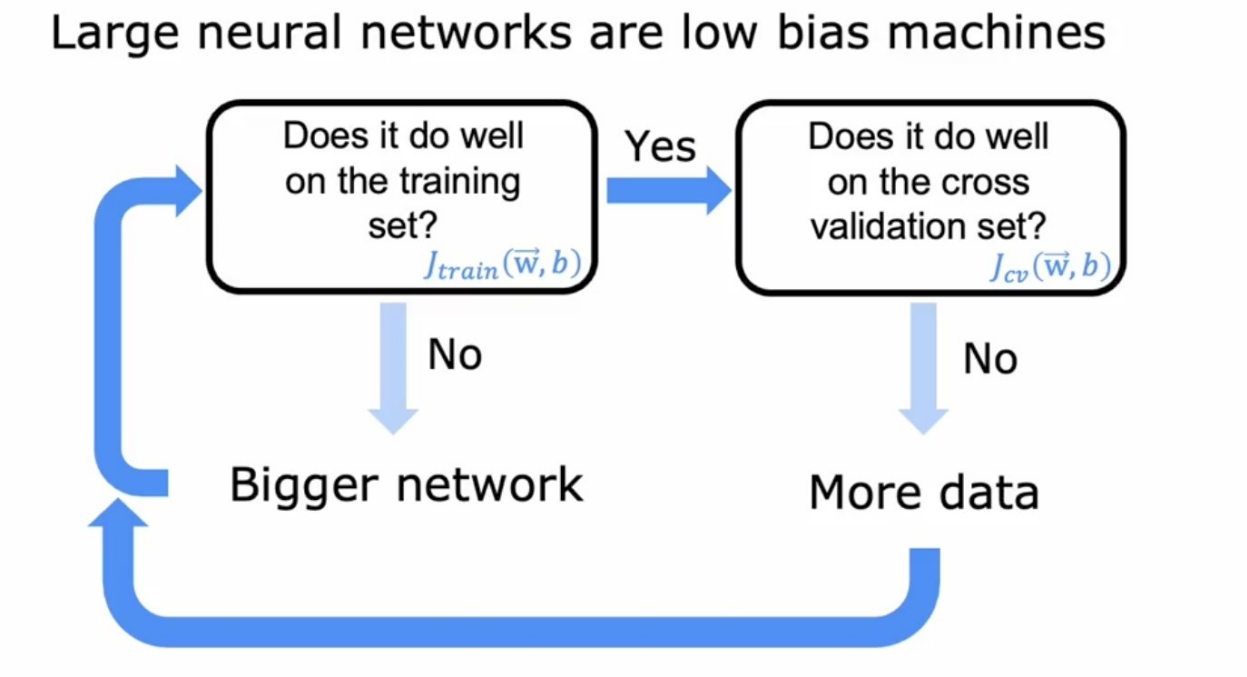
Debug a learning algorithm

- 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.



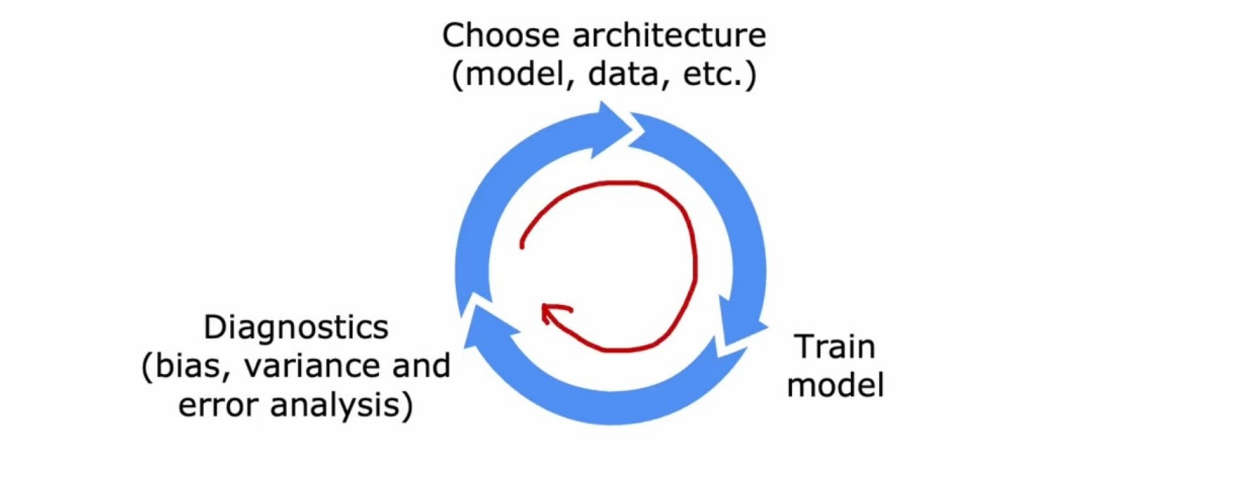

```
    layer_1 = Dense(units=25, activation='relu', kernel_regularizer=L2(0.01))
    layer_2 = Dense(units=15, activation='relu', kernel_regularizer=L2(0.01))
```

```
    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

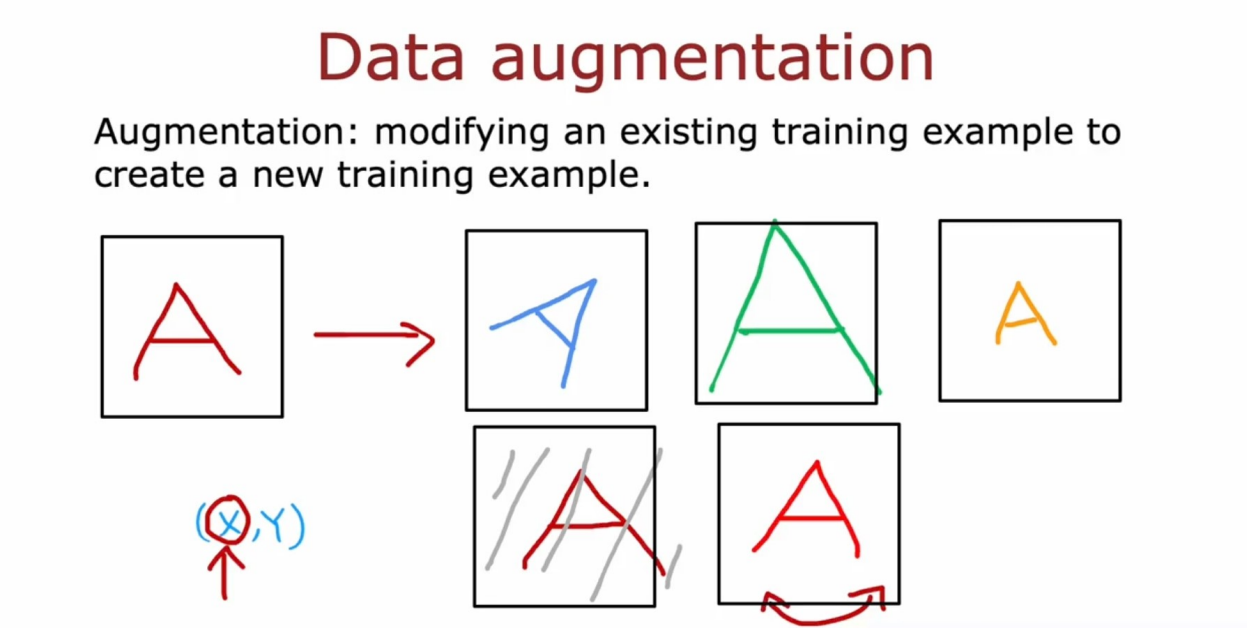


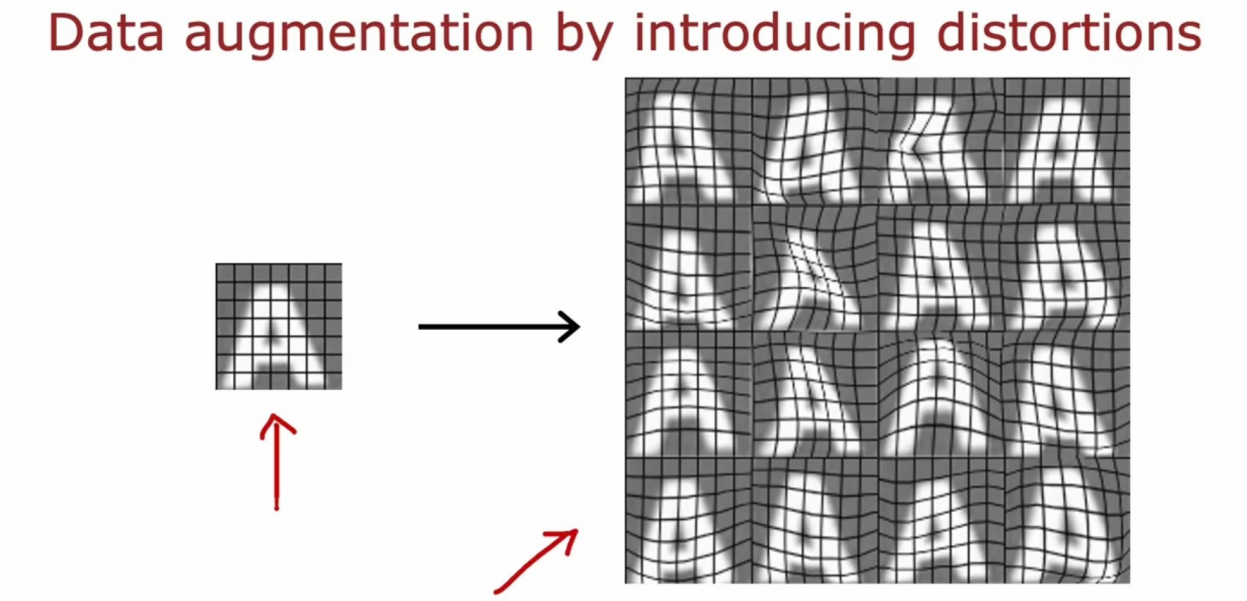
Error analysis

- 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?")

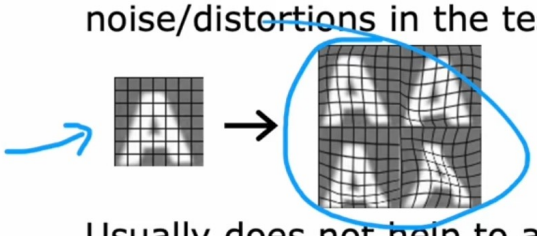
\+ Noisy background: Crowd

\+ Noisy background: Car

\+ Audio on bad cellphone connection

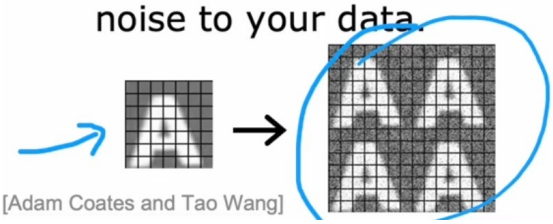
Data augmentation by introducing distortions

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



Audio:
Background noise,
bad cellphone connection

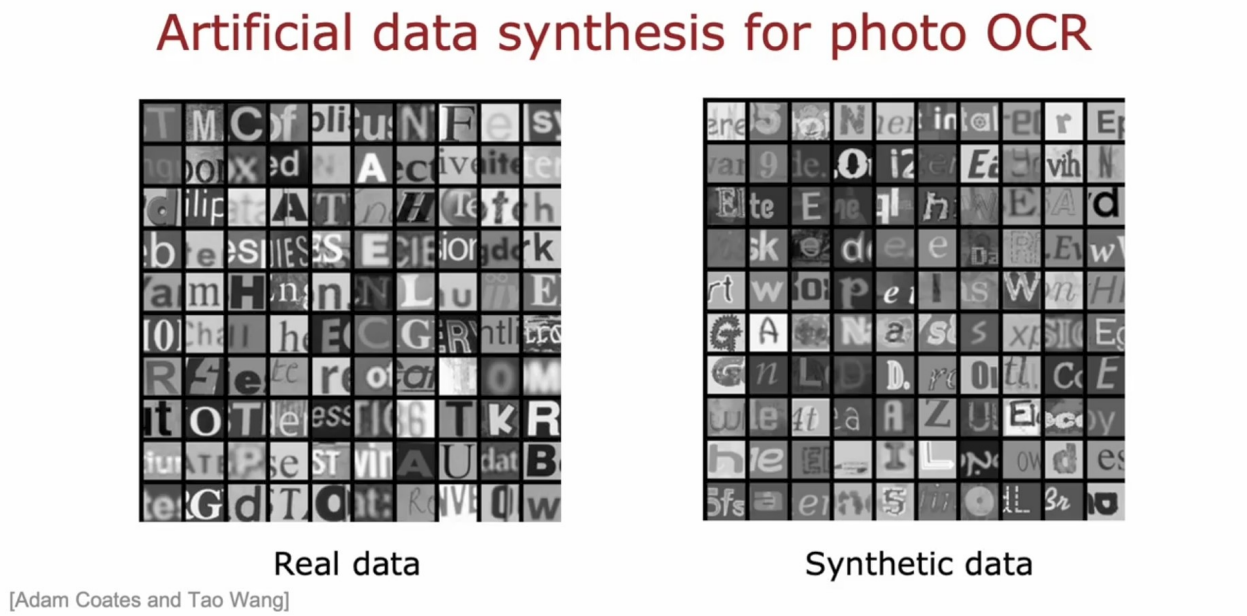
Usually does not help to add purely random/meaningless noise to your data.



[Adam Coates and Tao Wang]

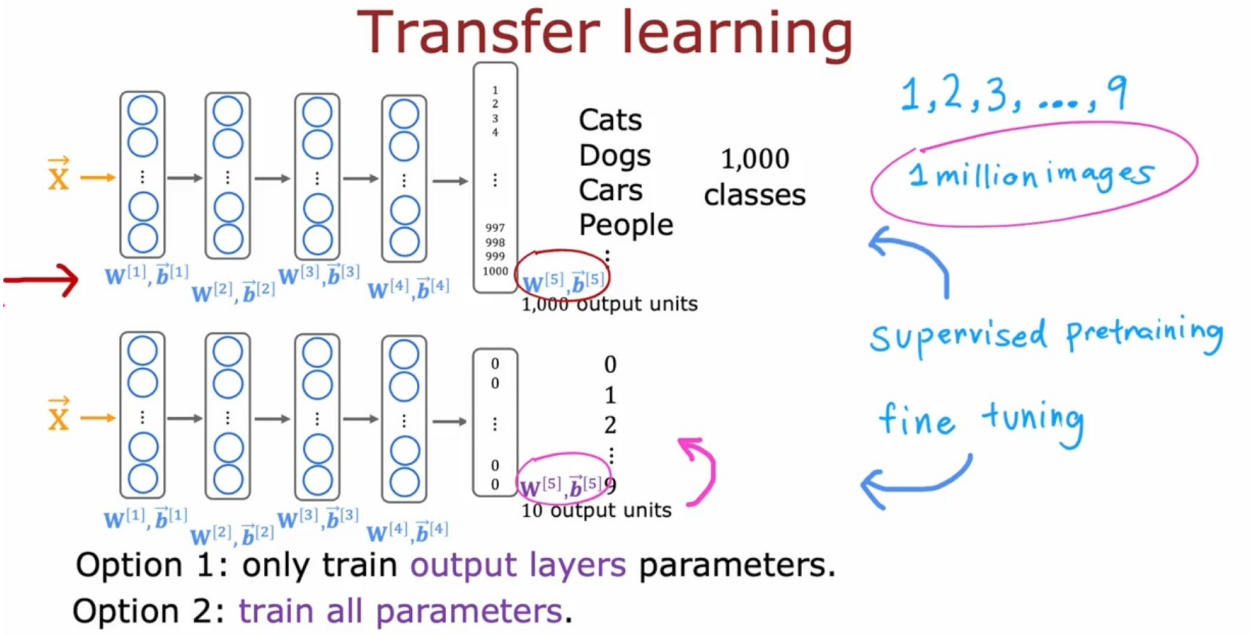
x_i=intensity (brightness) of pixel i

$x_i \leftarrow x_i$ + random noise



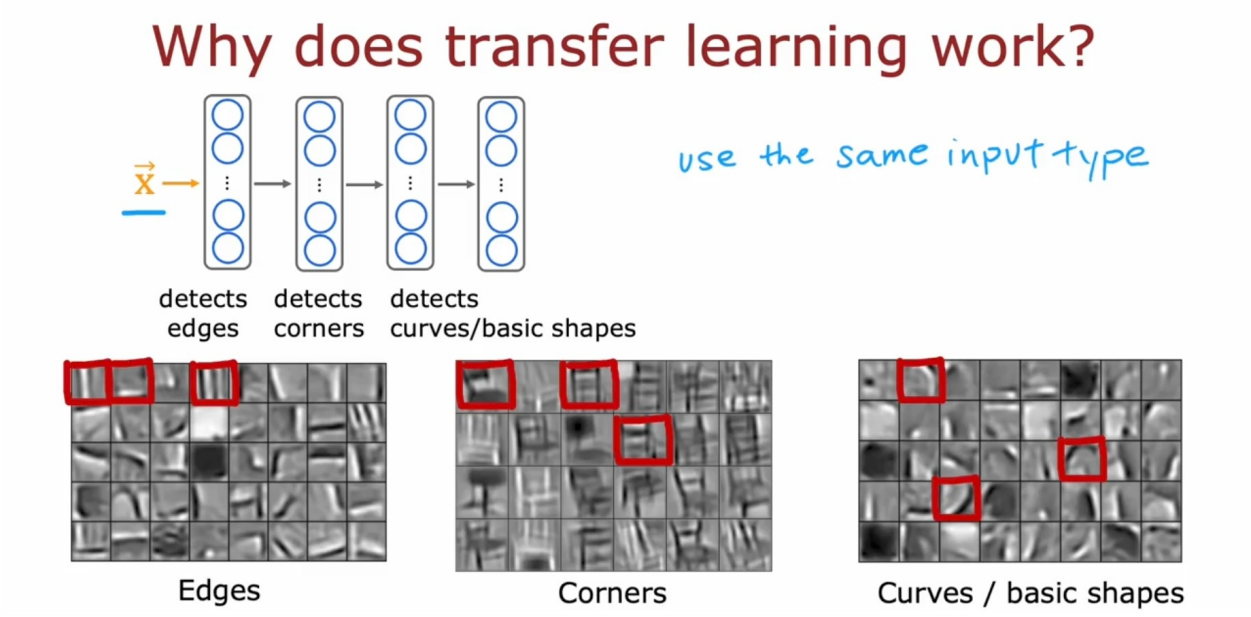
Transfer learning

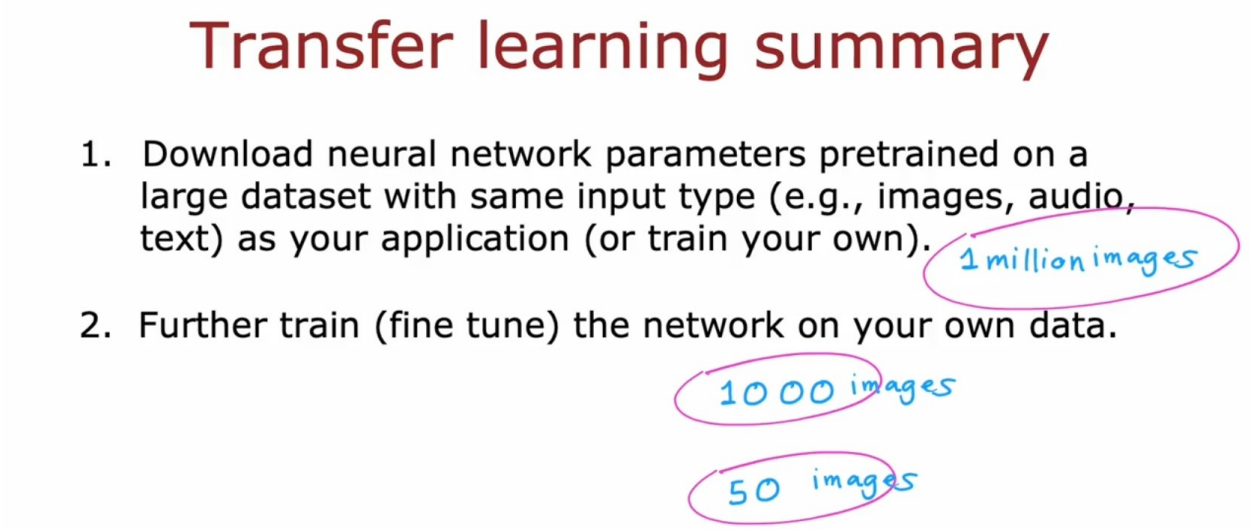
Transfer learning has 2 steps: **supervised pretraining** and **fine-tuning**. We can use a pre-trained model and fine-tune it to our problem. To do it, we can only change the output layer of the model.



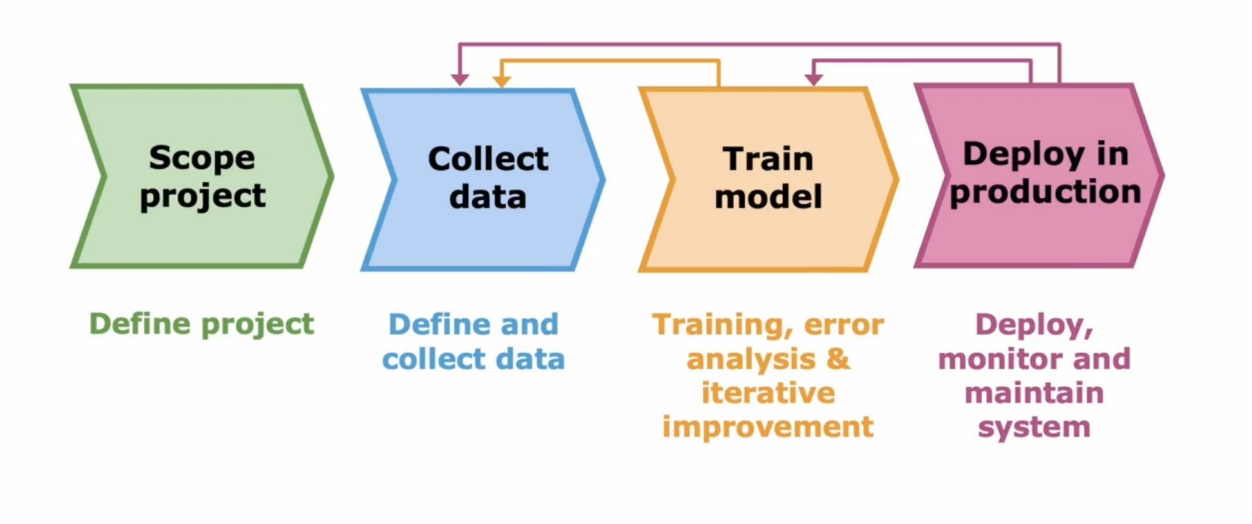
The reason for that is when input are of the same type, the first few layers of the neural network will learn features that are useful for the new task. Taking the CV model as an example, the first few layers will learn features that are general to all images such as edges, shapes,

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

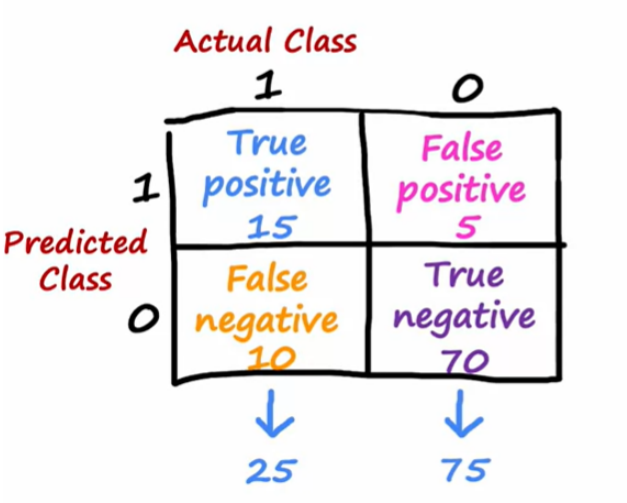
Error metrics for skewed datasets

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.

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

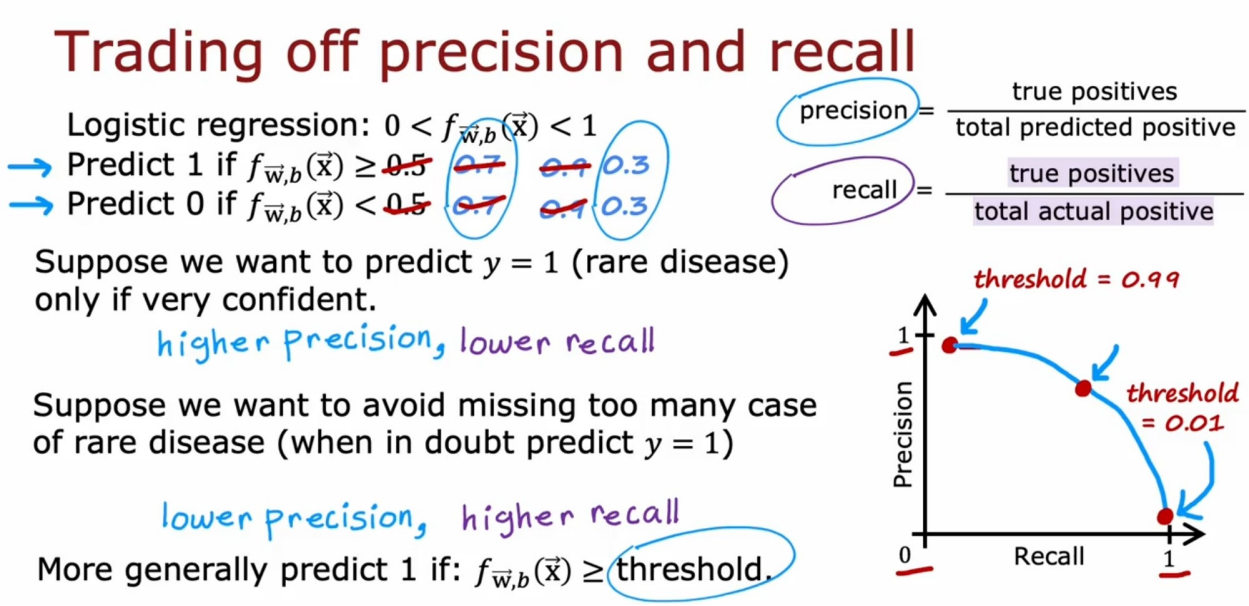
- Recall: the fraction of the positive examples that the model correctly predicts.

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

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.



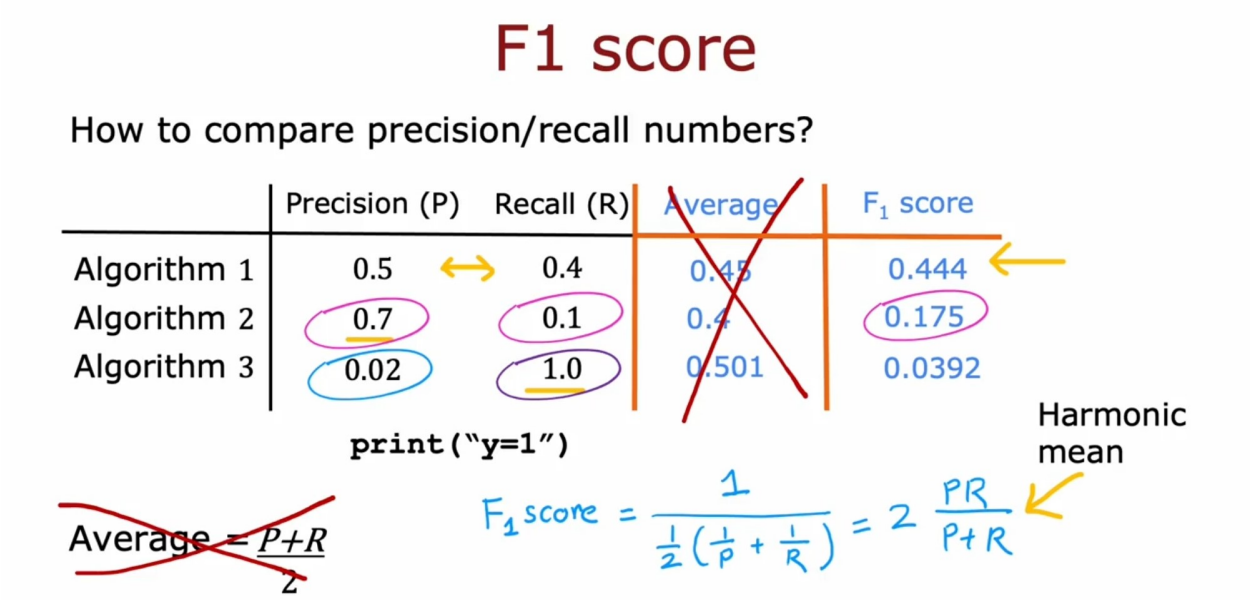
F1 score

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

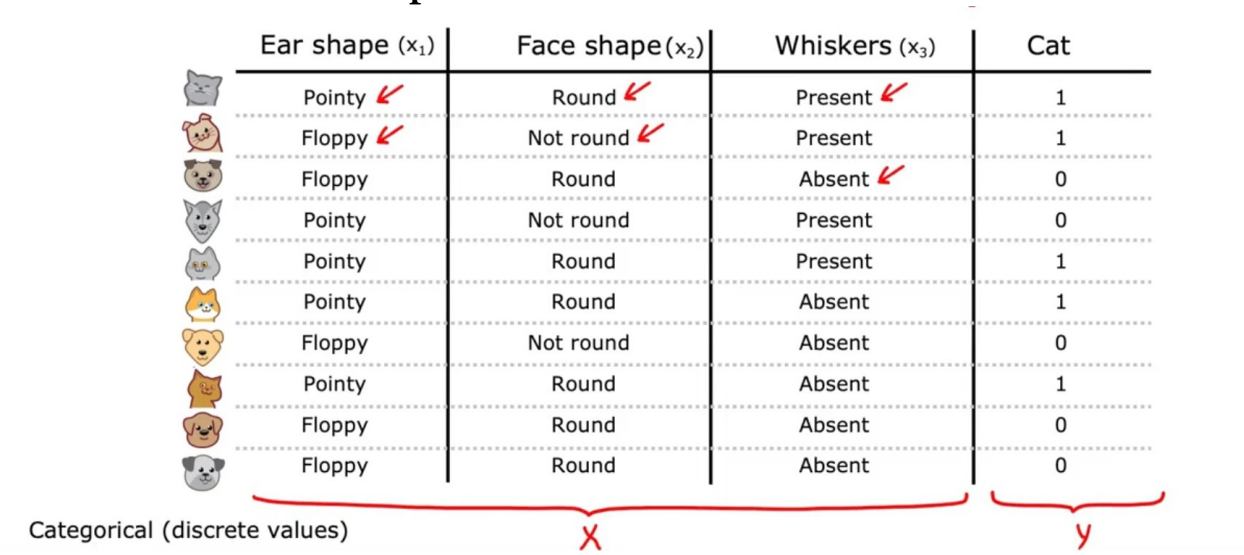
11

Decision Trees

11.1 Introduction

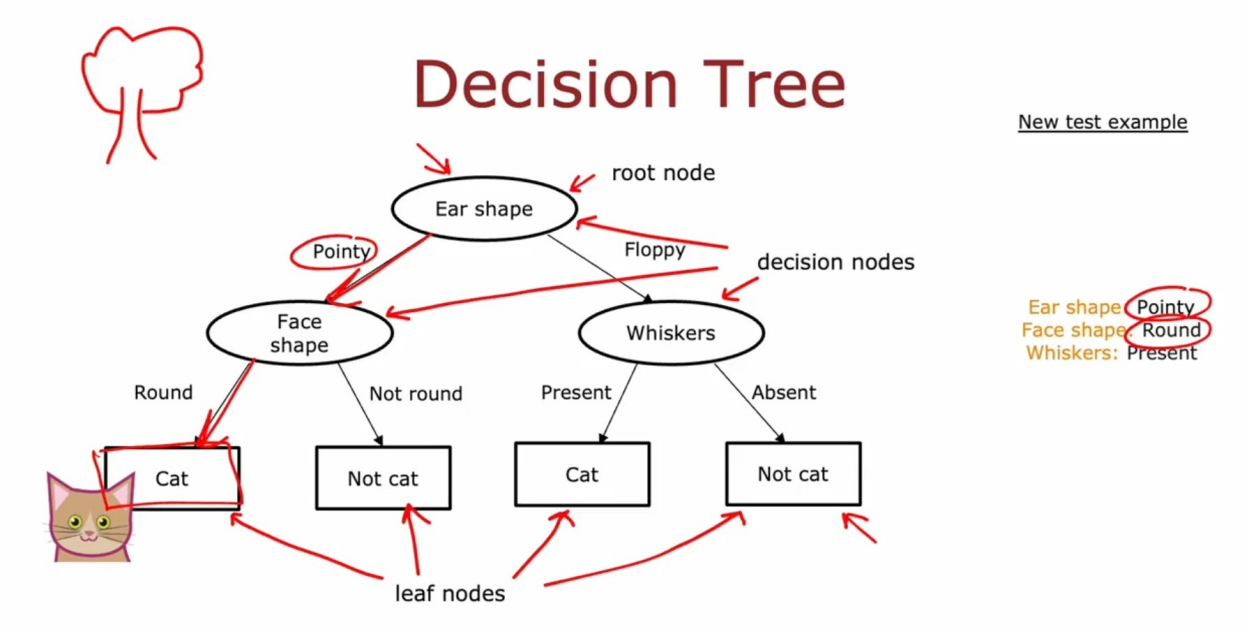
Decision trees are a popular method for various machine learning tasks. They are easy to understand and interpret, and the process of building a decision tree is intuitive.

Cats 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

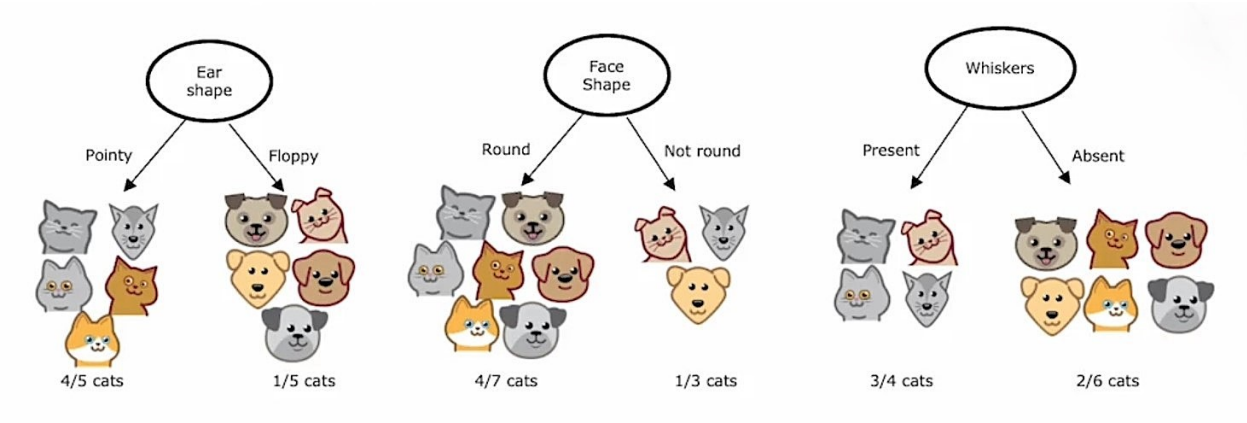
Decision tree model has the structure of a tree, where each **decision node** represents a feature (attribute), and each **leaf node** represents the outcome.



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:

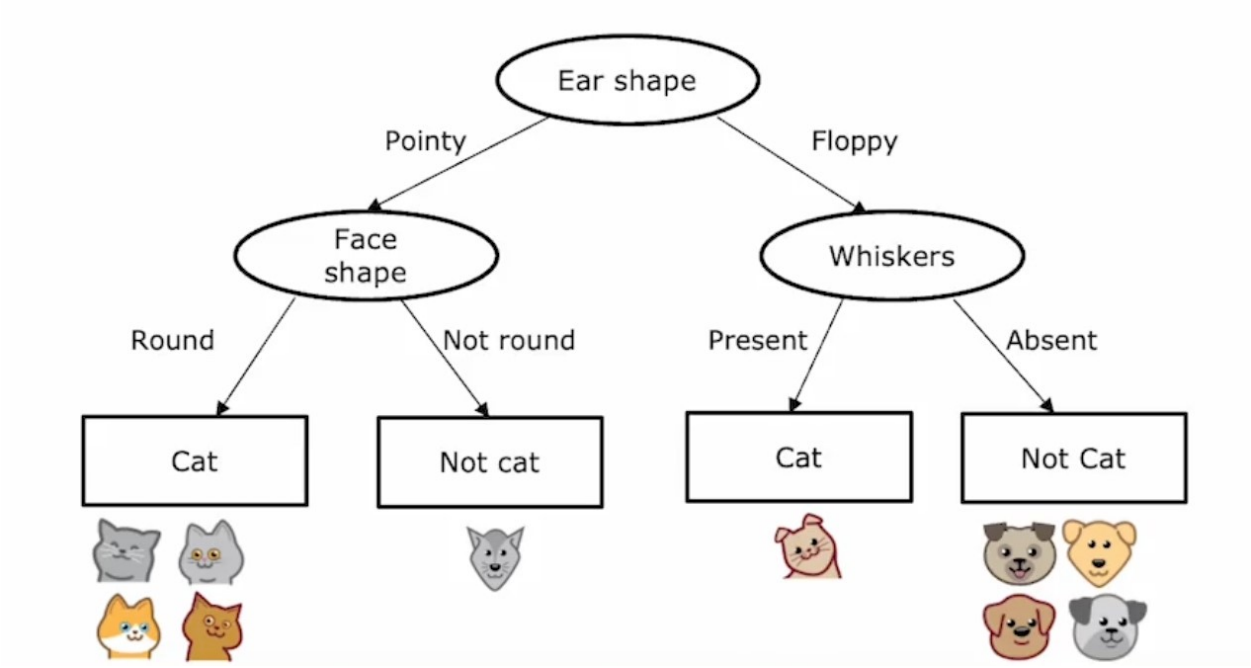
Decision 1: How to choose what features to split on at each node?
maximize purity (minimize impurity) of the child nodes.



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.



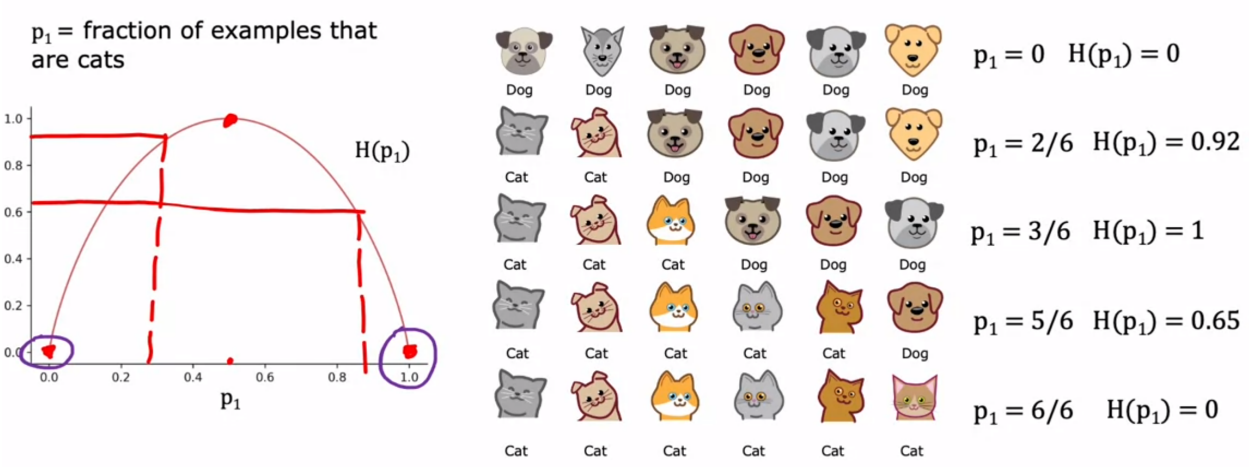
The splitting method and stopping method have a lots of variance, as the development of decision tree algorithm, many methods have been proposed.

11.2 Decision tree learning

Measuring purity

Entropy

Entropy is a measure of impurity.



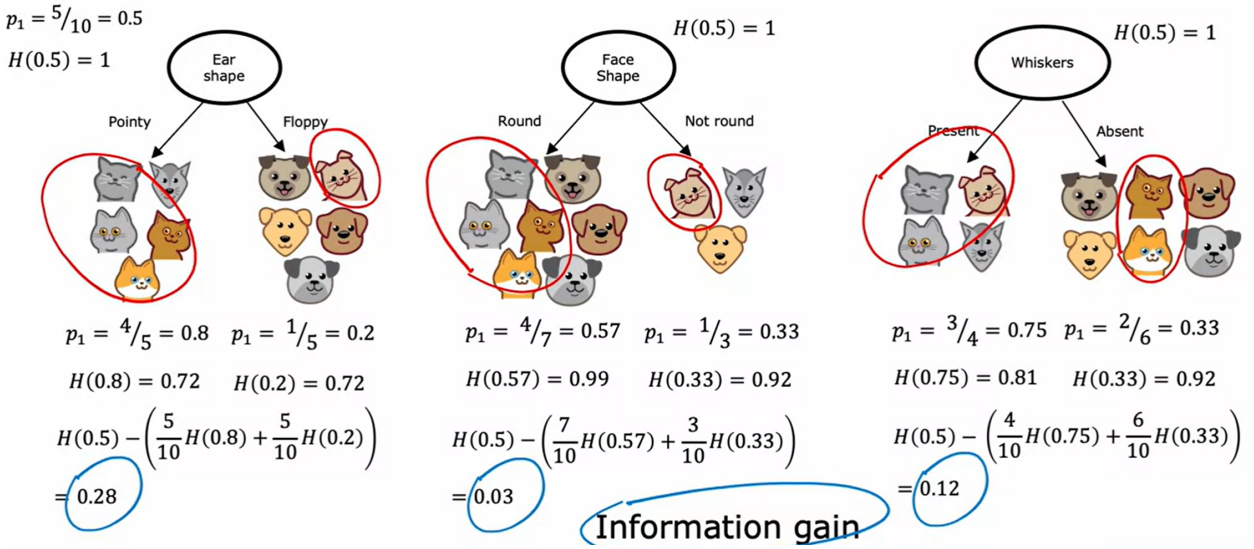
Theorem 11.2.1 ▸ Entropy

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

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

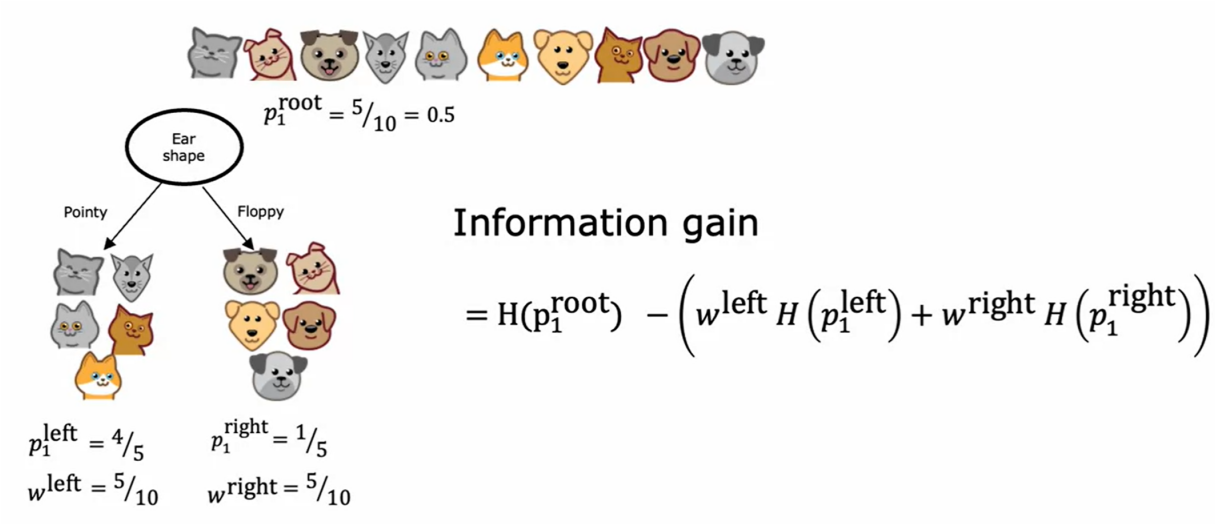
Information gain

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. `current node = root node`
2. `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
```

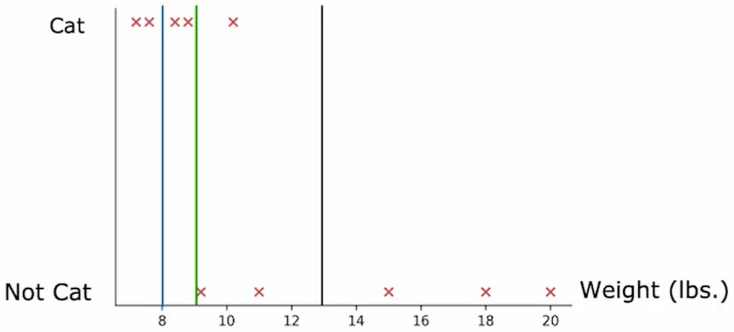
One-hot encoding

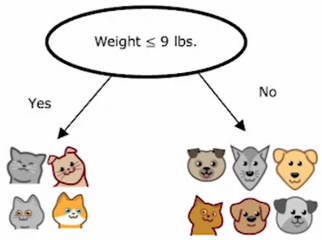
One-hot encoding is a method to convert categorical data into numerical data. if a feature has n categories, then it will be converted into n binary features.

	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.





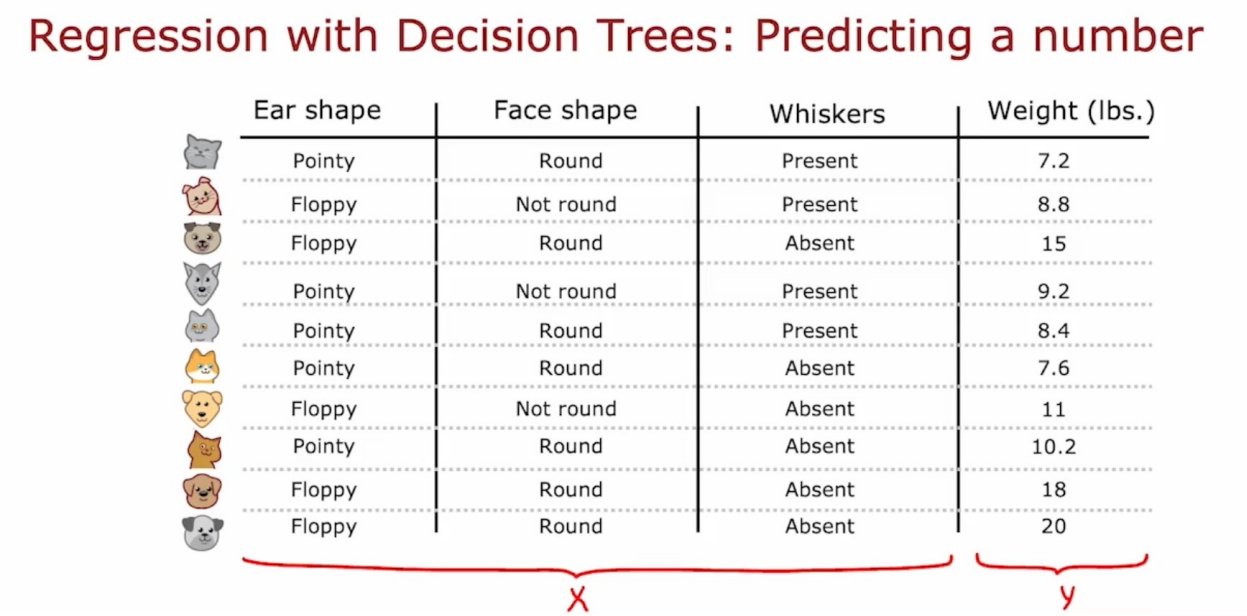
$$H(0.5) - \left(\frac{2}{10}H\left(\frac{2}{2}\right) + \frac{8}{10}H\left(\frac{3}{8}\right)\right) = 0.24$$

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

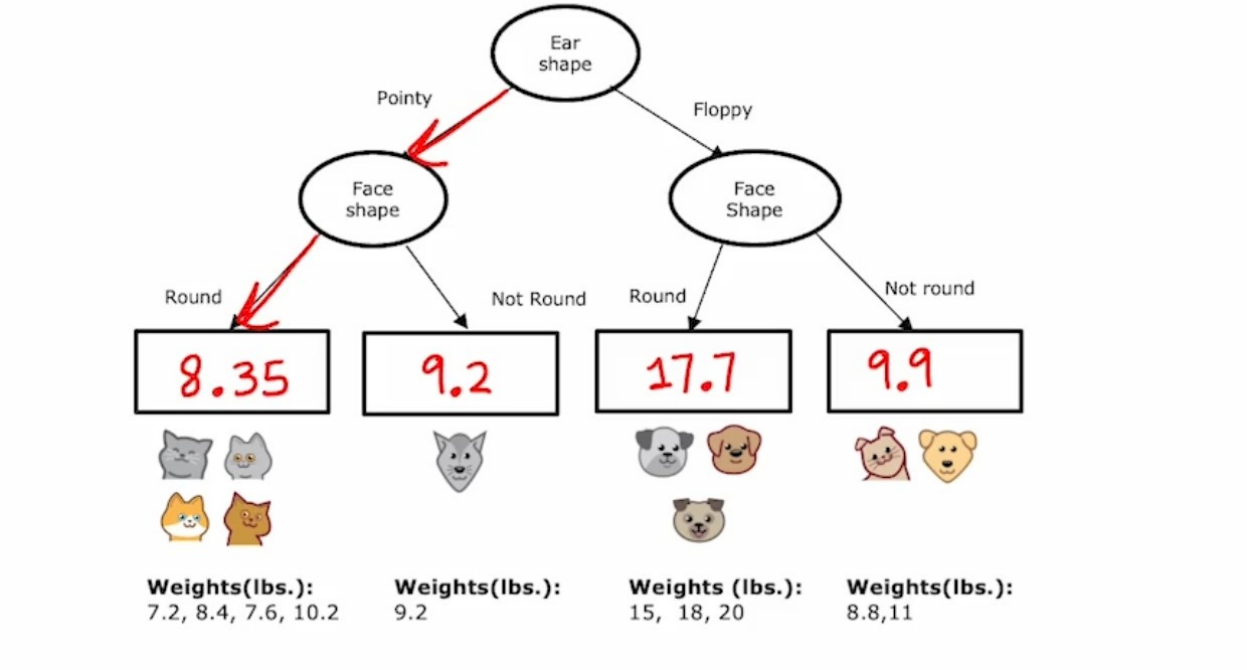
$$H(0.5) - \left(\frac{7}{10}H\left(\frac{5}{7}\right) + \frac{3}{10}H\left(\frac{0}{3}\right)\right) = 0.40$$

Regression trees

Regression trees are used for continuous target variables.

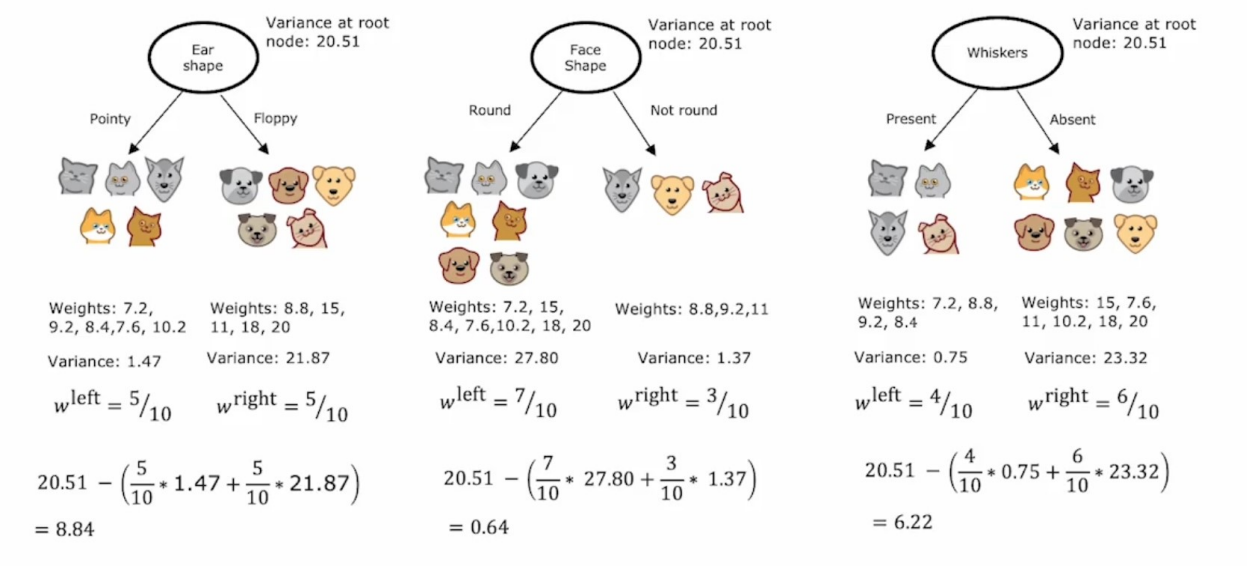


Assume that we have built a decision tree model, and we want to predict the target value for a new data point. The prediction is made by traversing the tree from the root node to a leaf node. The target value of the leaf node (average of target values of the training data in that node) is the prediction.



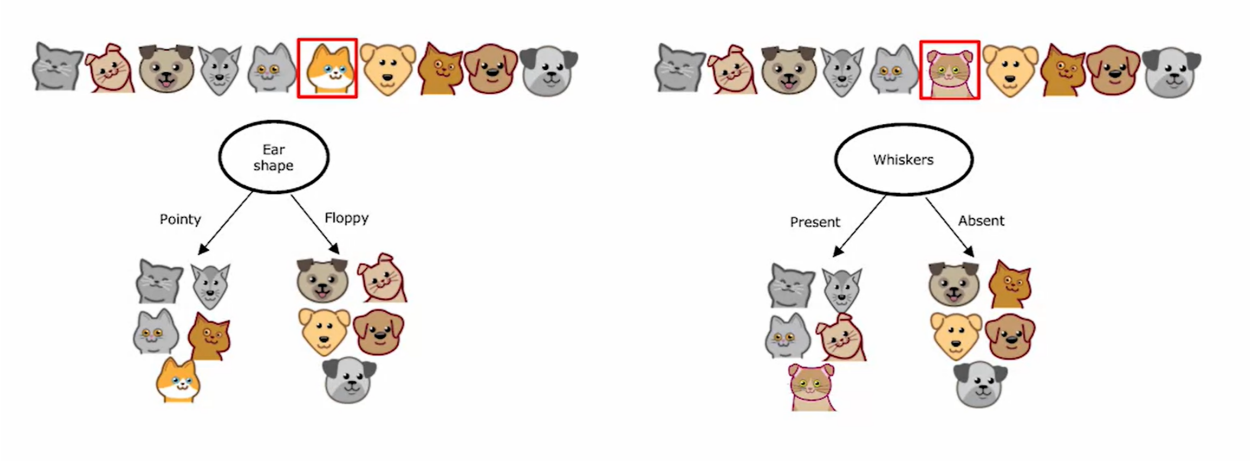
Choosing a split:

- 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.

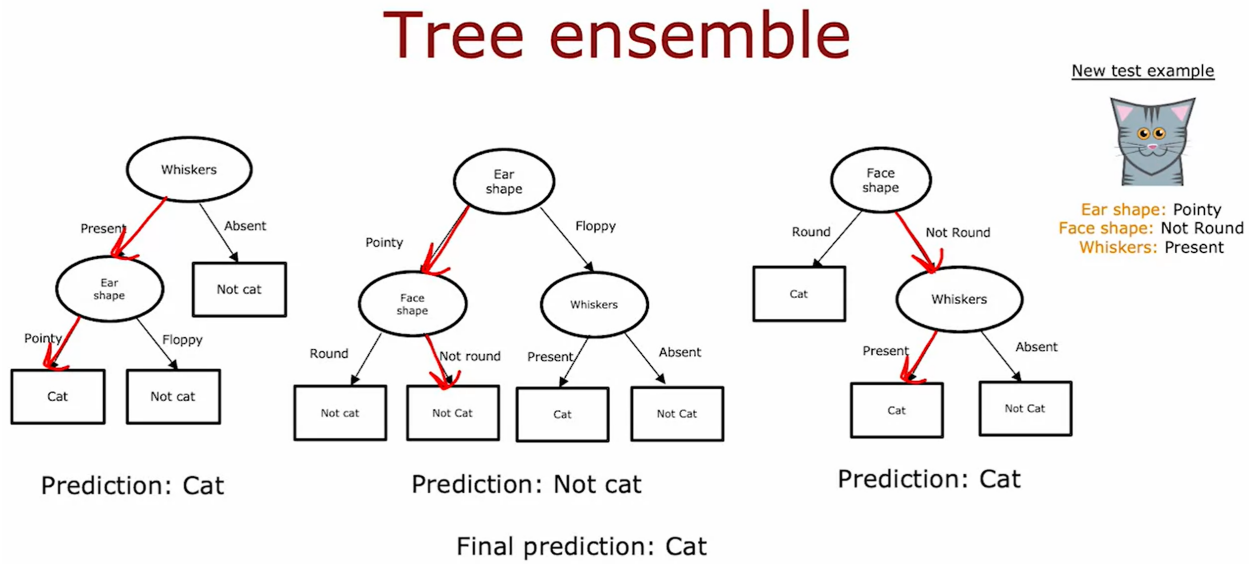


11.3 Tree ensembles

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.



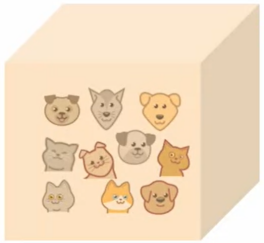
Therefore, we can use tree ensembles to improve the model's performance. Tree ensembles are a collection of decision trees that work together to make predictions. The process is that each tree makes a prediction, and the final prediction is the most voted prediction (or the average of all trees' predictions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

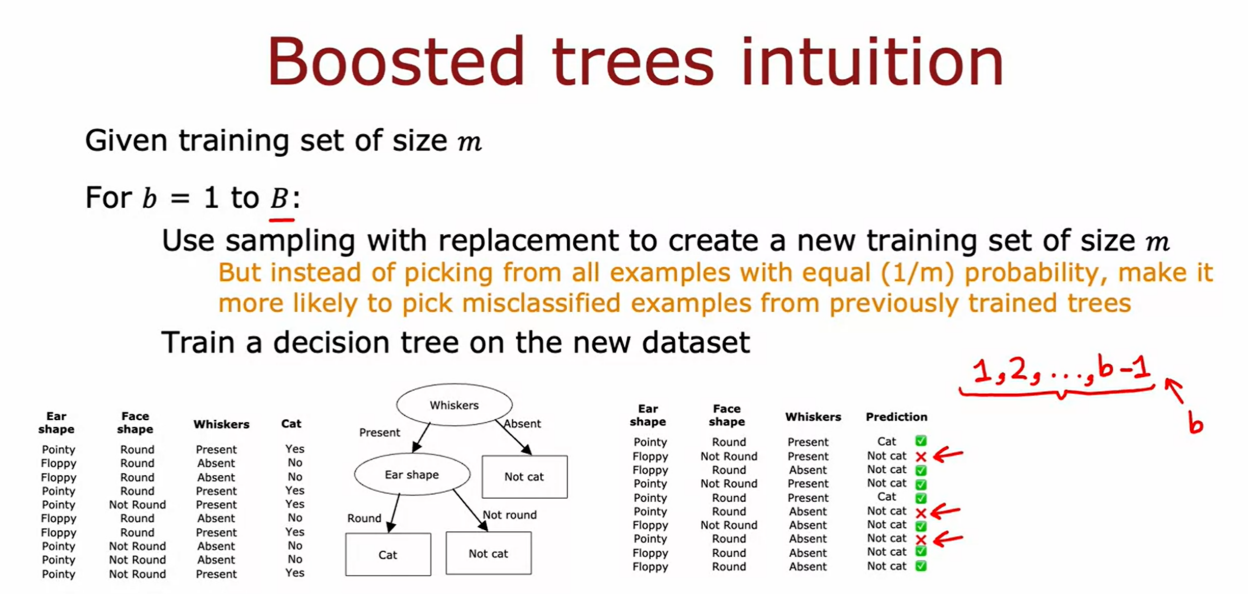
So the random forest algorithm is a bagged decision tree with randomizing the features.

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

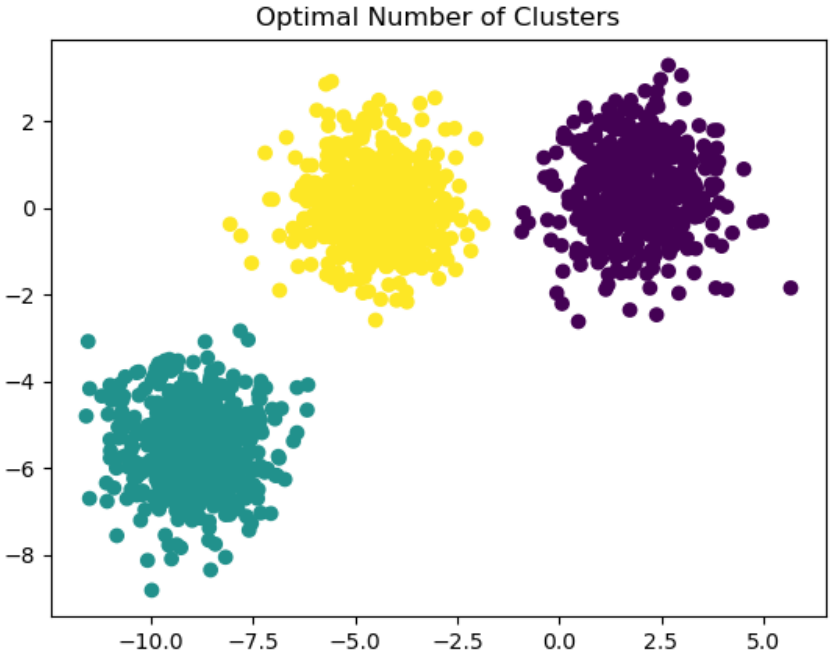
Unsupervised Learning, Recommender Systems and Reinforcement Learning

12 Clustering

12.1 What is clustering?

Clustering is the task of dividing the population or data points into a number of groups such that data points in the same groups are more similar to other data points in the same group and dissimilar to the data points in other groups. It is basically a collection of objects on the basis of similarity and dissimilarity between them.

Usually, we can represent the training set as $\{x^{(1)}, x^{(2)}, x^{(3)}, \cdots, x^{(m)}\}$, where $x^{(i)}$ is a feature vector representing the i^{th} training example. For example if the number of features n is 2, then $x^{(i)}$ is a 2D vector. We can draw the scatter plot of the training set, and we can see that the training set is divided into several clusters.



12.2 K-means Algorithm

K-means Intuition

The K-means algorithm is a method to automatically cluster similar data examples together. Here is the steps of the K-means algorithm:

Theorem 12.2.1 ▶ Simplified K-means Algorithm

step 1 : Assign each data point to the closest centroid.

step 2 : Compute the new centroids by averaging the data points assigned to each centroid.

Algorithm

Theorem 12.2.2 ▶ K-means Algorithms

1. Randomly initialize the K cluster centroids $\mu_1, \mu_2, \cdots, \mu_K$.
2. Repeat {
   for $i = 1$ to m {
   $c^{(i)}$:= index of cluster centroid closest to $x^{(i)}$
   }
   for $k = 1$ to K {
   μ_k := average of points in cluster k
   }
   }

- K is the number of clusters
- m is the number of training examples
- n is the number of features
- $c^{(i)}$ is the index of the centroid that is closest to $x^{(i)}$
- μ_k is the location of the k^{th} centroid

If one cluster centroid is not assigned any data points, then usually let K = K - 1 and repeat the algorithm. But if it is requested to keep K clusters, then you can randomly initialize the centroid again.

12.3 Optimization Objective

Theorem 12.3.1 ▸ Cost function of K-means

- $c^{(i)}$:= index of cluster centroid which example $x_{(i)}$ is currently assigned to.
- μ_k := cluster centroid k.
- $\mu_{c^{(i)}}$:= cluster centroid of cluster to which example $x^{(i)}$ has been assigned.

the μ_k and $\mu c^{(i)}$ are both n-dimensional vectors.

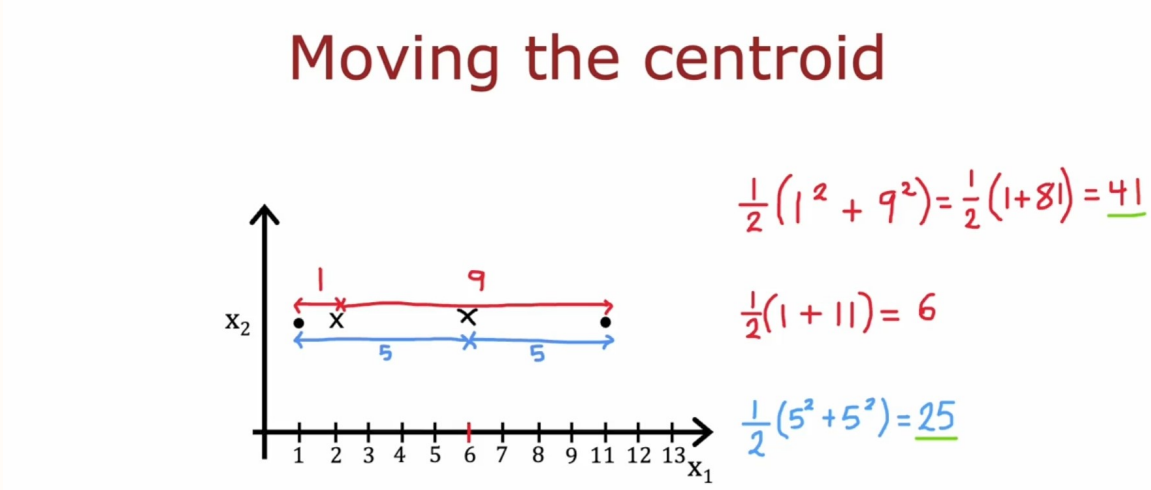
$$J(c^{(1)}, \cdots, c^{(m)}, \mu_1, \cdots, \mu_K) = \frac{1}{m} \sum_{i=1}^{m} \|x^{(i)} - \mu_{c^{(i)}}\|^2 \tag{12.1}$$

Our goal is to find the $c^{(1)}, \cdots, c^{(m)}, \mu_1, \cdots, \mu_K$ to minimize J.

According to the K-means algorithm, first step is to assign each data point to the closest centroid. In this procedure, we minimize J with respect to $c^{(1)}, \cdots, c^{(m)}$ while remains the $\mu_1, \cdots, \mu_K$ fixed.

The second step is to compute the new centroids by averaging the data points assigned to each centroid. In this procedure, we minimize J with respect to $\mu_1, \cdots, \mu_K$, while remains the $c^{(1)}, \cdots, c^{(m)}$ fixed.

Example 12.3.2 ▸ Moving the centroid



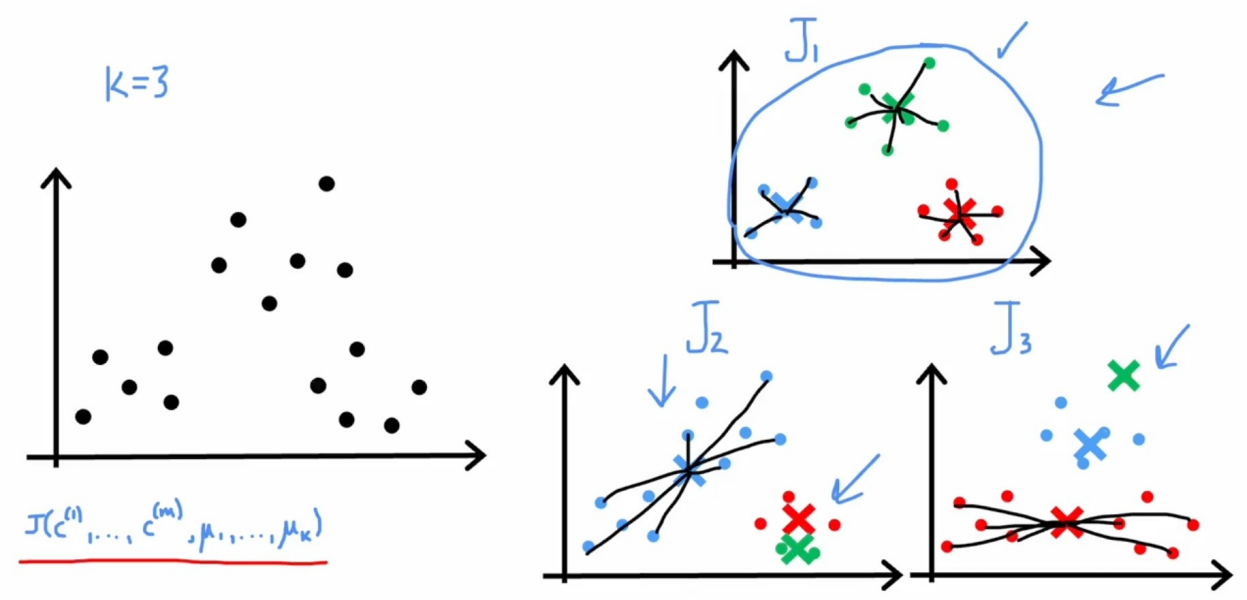
The K-means algorithm is guaranteed to converge to a local optimum. And after each

iteration, the cost function J will always decrease. But the K-means algorithm may not converge to the global optimum. Even delt with not well-separated clusters, we can also apply the K-means algorithm to separate them.

12.4 Random Initialization

1. Choose $K < m$.
2. Randomly pick K training examples.
3. Set $\mu_1, \mu_2, \cdots, \mu_K$ equal to these K examples.

- The K-means algorithm can converge to different solutions depending on the initial setting of the centroids.
- To solve this problem, we can run the K-means algorithm multiple times with different random initializations.
- We can then choose the clustering that gave us the lowest cost.



Theorem 12.4.1 ▸ Random Initialization

For $i = 1$ to 100 {
  Randomly initialize K-means.
  Run K-means. Get $c^{(1)}, \cdots, c^{(m)}, \mu_1, \cdots, \mu_K$.
  Compute cost function (distortion)$J(c^{(1)}, \cdots, c^{(m)}, \mu_1, \cdots, \mu_K)$.

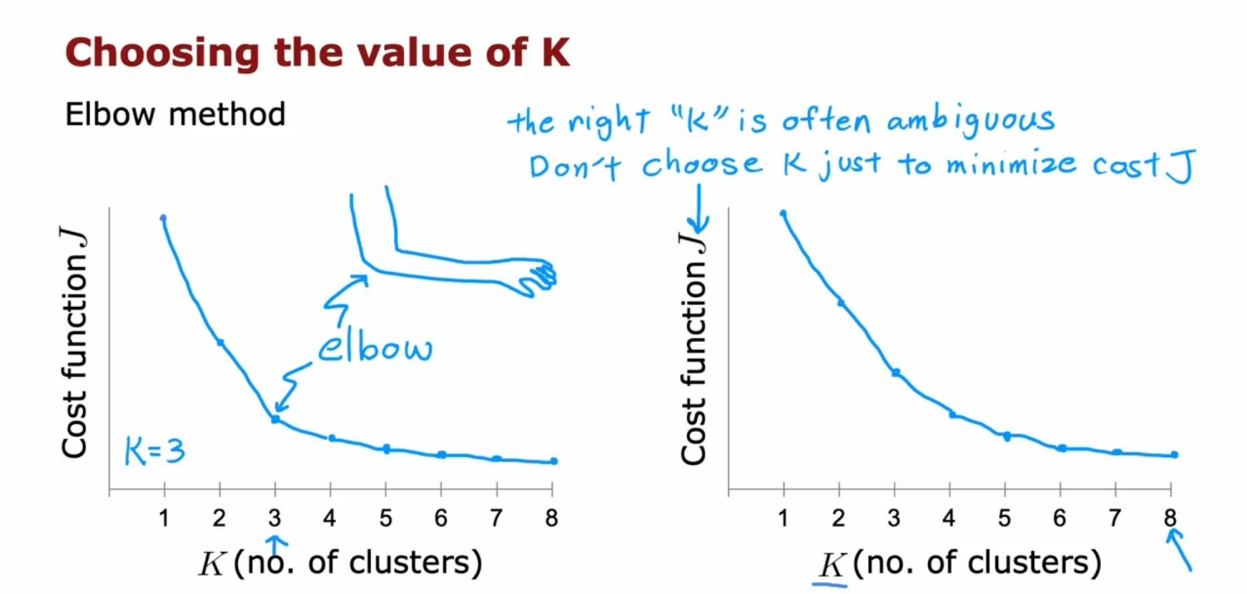
}
Pick clustering that gave lowest cost J.

The training times is often set to 50 - 1000.

12.5 Choosing the Number of Clusters

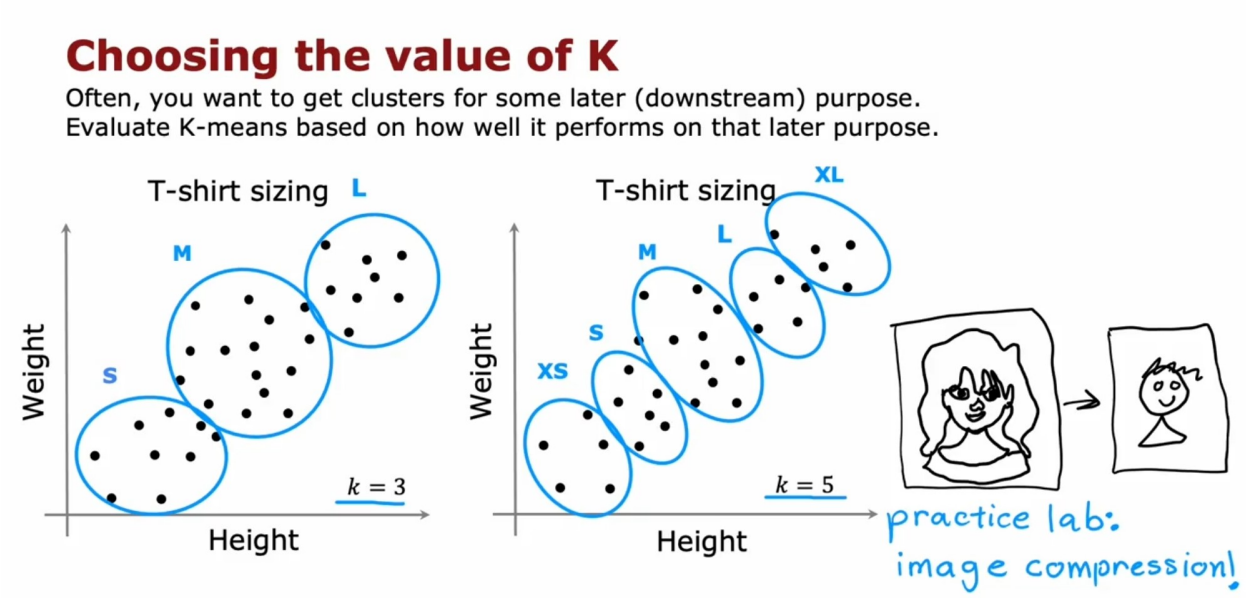
Elbow Method

1. Run K-means for a range of values of K.
2. For each value of K, calculate the distortion.
3. Plot the distortion against the number of clusters K.
4. The elbow point is the point where the distortion starts to decrease more slowly.



The elbow point is not clear most of the time.

And don't always need to choose the number of clusters that minimize the distortion. Because as the number of clusters increases, the distortion will decrease all the time.



12.6 K-means from Scratch

file name: kmeans.py

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.datasets import load_wine

def find_closest_centroids(X, centroids):
    """
    Computes the centroid memberships for every example

    Args:
        X (ndarray): (m, n) Input values
        centroids (ndarray): k centroids

    Returns:
        idx (array_like): (m,) closest centroids
    """
    # Set K
    K = centroids.shape[0]
```

```
    # You need to return the following variables correctly
    idx = np.zeros(X.shape[0], dtype=int)

    for i in range(X.shape[0]):
        # Array to hold distance between X[i] and each centroids[j]
        distance = []
        for j in range(K):
            norm_ij = np.linalg.norm(X[i] - centroids[j])
            distance.append(norm_ij)
        idx[i] = np.argmin(distance)

    return idx

def compute_centroids(X, idx, K):
    """
    Returns the new centroids by computing the means of the
    data points assigned to each centroid.

    Args:
        X (ndarray):   (m, n) Data points
        idx (ndarray): (m,) Array containing index of closest centroid for
         ↪  each
                    example in X. Concretely, idx[i] contains the index of
                      the centroid closest to example i
        K (int):       number of centroids

    Returns:
        centroids (ndarray): (K, n) New centroids computed
    """
    # Useful variables
    m, n = X.shape

    # You need to return the following variables correctly
    centroids = np.zeros((K, n))
```

```
    for k in range(K):
          points = X[idx == k, :]
          centroids[k] = np.mean(points, axis = 0)

    return centroids

def run_kMeans(X, initial_centroids, max_iters=10):
    """
    Runs the K-Means algorithm on data matrix X, where each row of X
    is a single example

    Args:
        X (ndarray): (m, n) Data points
        initial_centroids (ndarray): (K, n) Initial centroids
        max_iters (int): number of iterations to run
        plot_progress (bool): True to plot progress, False otherwise

    Returns:
        centroids (ndarray): (K, n) Final centroids
        idx (ndarray): (m,) Index of the closest centroid for each example
    """
    # Initialize values
    m, n = X.shape
    K = initial_centroids.shape[0]
    centroids = initial_centroids
    idx = np.zeros(m)

    # Run K-Means
    for i in range(max_iters):

        # For each example in X, assign it to the closest centroid
        idx = find_closest_centroids(X, centroids)

        # Given the memberships, compute new centroids
```

```
        centroids = compute_centroids(X, idx, K)

    return centroids, idx

def kMeans_init_centroids(X, K):
    """
    This function initializes K centroids that are to be
    used in K-Means on the dataset X

    Args:
        X (ndarray): Data points
        K (int):     number of centroids/clusters

    Returns:
        centroids (ndarray): Initialized centroids
    """
    # Randomly reorder the indices of examples
    randidx = np.random.permutation(X.shape[0])

    # Take the first K examples as centroids
    centroids = X[randidx[:K]]

    return centroids

def visualize_kMeans(X, centroids, idx):
    """
    Plots the data points with colors assigned to each centroid.

    Args:
        X (ndarray): (m, n) Data points
        centroids (ndarray): (K, n) Centroids
        idx (ndarray): (m,) Index of the closest centroid for each example
    """
    # Useful variables
    K = centroids.shape[0]
```

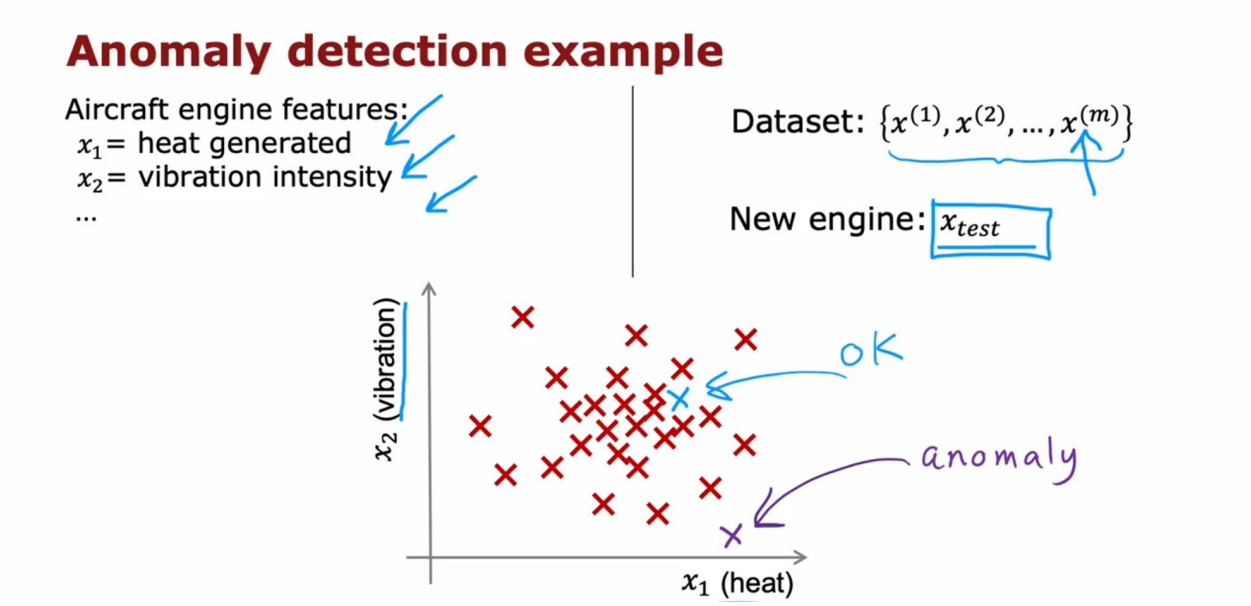
```
    m, n = X.shape

    # pca for dimensionality reduction
    pca = PCA(n_components=2)
    X = pca.fit_transform(X)
    centroids = pca.transform(centroids)

    # Plot the data
    plt.scatter(X[:, 0], X[:, 1], c=idx, cmap='viridis', marker='o')
    plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='x',
     ↪  s=100)
    plt.show()

def run_example():
    # Load an example dataset
    data = load_wine()
    X = data.data
    y = data.target

    # Settings for running K-Means
    K = 3
    max_iters = 10

    # For consistency, here we set centroids to specific values
    initial_centroids = kMeans_init_centroids(X, K)

    # Run K-Means algorithm
    centroids, idx = run_kMeans(X, initial_centroids, max_iters)

    # Visualize the K-Means result
    visualize_kMeans(X, centroids, idx)


if __name__ == '__main__':
    run_example()
```

13 Anomaly Detection

13.1 Find unusual events

Definition 13.1.1 ▸ Anomaly detection

Anomaly detection is the task of finding unusual events in the data.



Anomaly detection example

how often log in?
how many web pages visited?
transactions?
posts? typing speed?

Fraud detection:
- $x^{(i)}$ = features of user i's activities
- Model $p(x)$ from data.
- Identify unusual users by checking which have $p(x) < \varepsilon$

perform additional checks to identify real fraud vs. false alarms

Manufacturing:
$x^{(i)}$= features of product i

airplane engine
circuit board
smartphone

ratios

Monitoring computers in a data center:
$x^{(i)}$= features of machine i
- x_1 = memory use,
- x_2= number of disk accesses/sec,
- x_3 = CPU load,
- x_4= CPU load/network traffic.

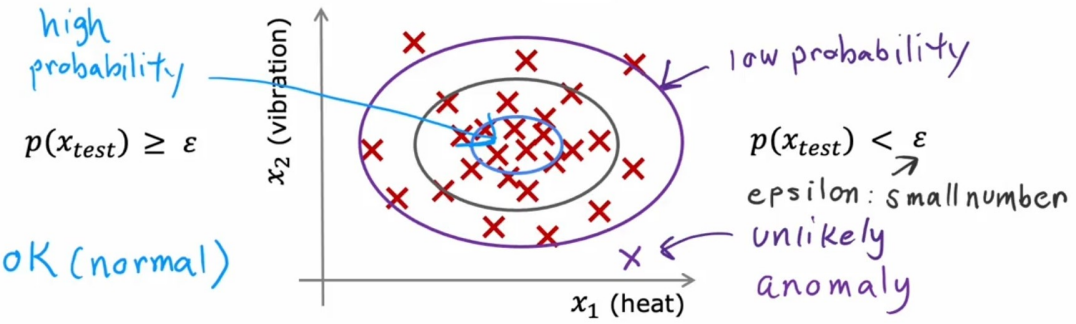
We will model the probability of the data, and flag data points with low probability as anomalies. we can set a threshold ε to determine which data points are anomalies. If $p(x) < \varepsilon$, then x is an anomaly.

Density estimation

Dataset: $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$ probability of x being seen in dataset

Model $p(x)$

Is x_{test} anomalous?



13.2 Algorithm

Gaussian Distribution

Contents about Gaussian distribution can be found in the Stanford CS109 course, so I will not repeat details here. But remeber that the PDF (Probability Density Function) of the Gaussian distribution is:

$$p(x;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{13.1}$$

where μ is the mean and σ^2 is the variance of the Gaussian distribution. Because the shape of the Gaussian distribution is like a bell, it is also called the normal distribution.

The parameters μ and σ^2 can be estimated from the data. So what we are going to do is compute the mean and variance of each feature in the training set. Below is the formal Anomaly Detection Algorithm:

Theorem 13.2.1 ▸ Anomaly Detection Algorithm

1. Choose n features x_i that you think might be indicative of anomalous examples.
2. fit parameters $\mu_1,\cdots,\mu_n,\sigma_1^2,\cdots,\sigma_n^2$:

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)} \tag{13.2}$$

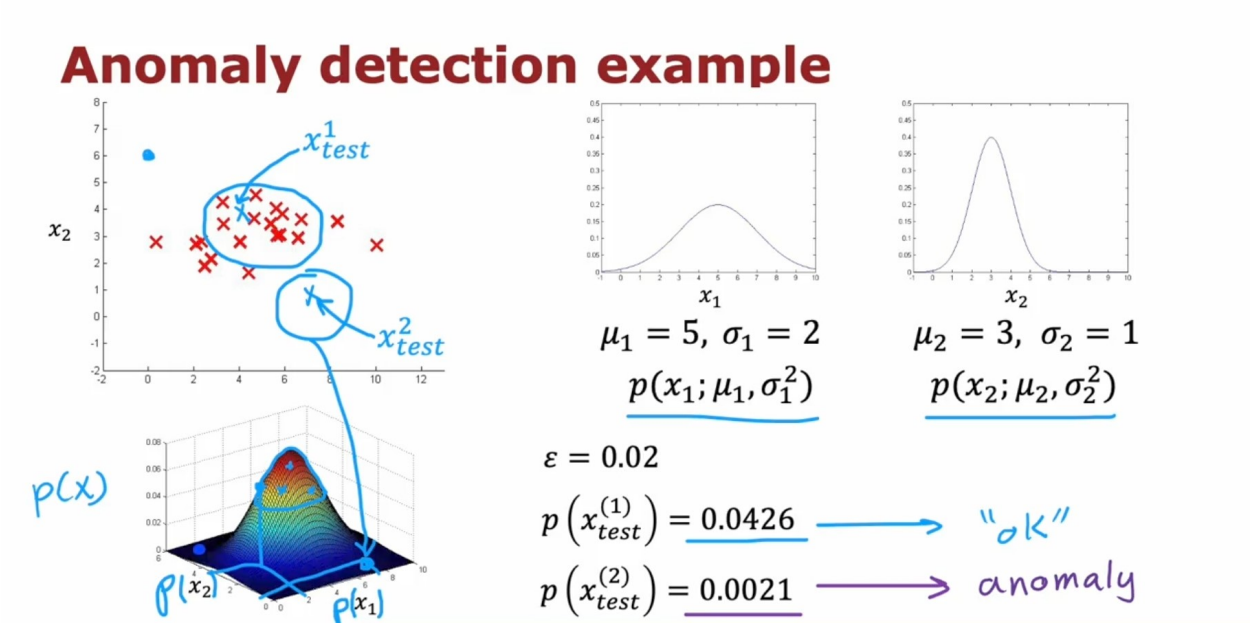
$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)}-\mu_j)^2 \tag{13.3}$$

3. Given a new example $\mathbf{x}$, compute $p(\mathbf{x})$:

$$p(\mathbf{x}) = \prod_{j=1}^{n} p(x_j;\mu_j,\sigma_j^2) = \prod_{j=1}^{n}\frac{1}{\sqrt{2\pi}\sigma_j}\exp\left(-\frac{(x_j-\mu_j)^2}{2\sigma_j^2}\right) \tag{13.4}$$

4. Anomaly if $p(\mathbf{x}) < \varepsilon$.

1. In some statistics books, you may see it devide by $m - 1$, but in machine learning, we usually devide by m. (This can be derived from the maximum likelihood estimation.)
2. the probability of the data point occurring can be calculated by multiplying the probabilities of each feature. So $p(\mathrm{x}) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2)$.



13.3 Developing and Evaluating an Anomaly Detection System

Evaluating algorithm

Assume that we have a labeled dataset, where the normal examples are labeled as $y = 0$ and the anomalous examples are labeled as $y = 1$.

We can devide the dataset into 3 parts: training set, cross validation set, and test set. The training set is used to fit the parameters, the cross validation set is used to choose the threshold ε, and the test set is used to evaluate the algorithm. And the training examples should all be normal examples, which means $y = 0$ (while it's okay if a few anomalous examples occur). The cross validation and test sets should have a small part of anomalous examples.

If in our datasets, there is a really small number of anomalous examples, it is recommended to drop the test set, and use all the anomalous examples in the cross validation set for choosing the threshold ε. While this method may lead to higher risk of overfitting, it is still a good way to evaluate the algorithm.

Theorem 13.3.1 ▶ Evaluation

1. Fit model $p(x)$ on training set $\{x^{(1)}, \cdots, x^{(m)}\}$.
2. On a cross validation/test example x, predict:

$$y = \begin{cases} 1 & \text{if } p(x) < \varepsilon \text{ (anomaly)} \\ 0 & \text{if } p(x) \geq \varepsilon \text{ (normal)} \end{cases} \tag{13.5}$$

3. Possible evaluation metrics:
   - True positive, false positive, false negative, true negative.
   - Precision/Recall.
   - F_1 score.
4. Use cross validation set to choose parameter ε.

> The "skewed datasets" method introduced in part 2 is commonly used in anomaly detection.

13.4 Anomaly Detection vs. Supervised Learning

Responding features:

- Very small number of positive examples ($y = 1$), Large number of negative examples ($y = 0$).
- Many different "types" of anomalies. Hard for any algorithm to learn from positive examples what the anomalies look like; future anomalies may look nothing like any of the anomalous examples we've seen so far.

- Large number of positive and negative examples.
- Enough positive examples for algorithm to get a sense of what positive examples are like, future positive examples likely to be similar to ones in training set.

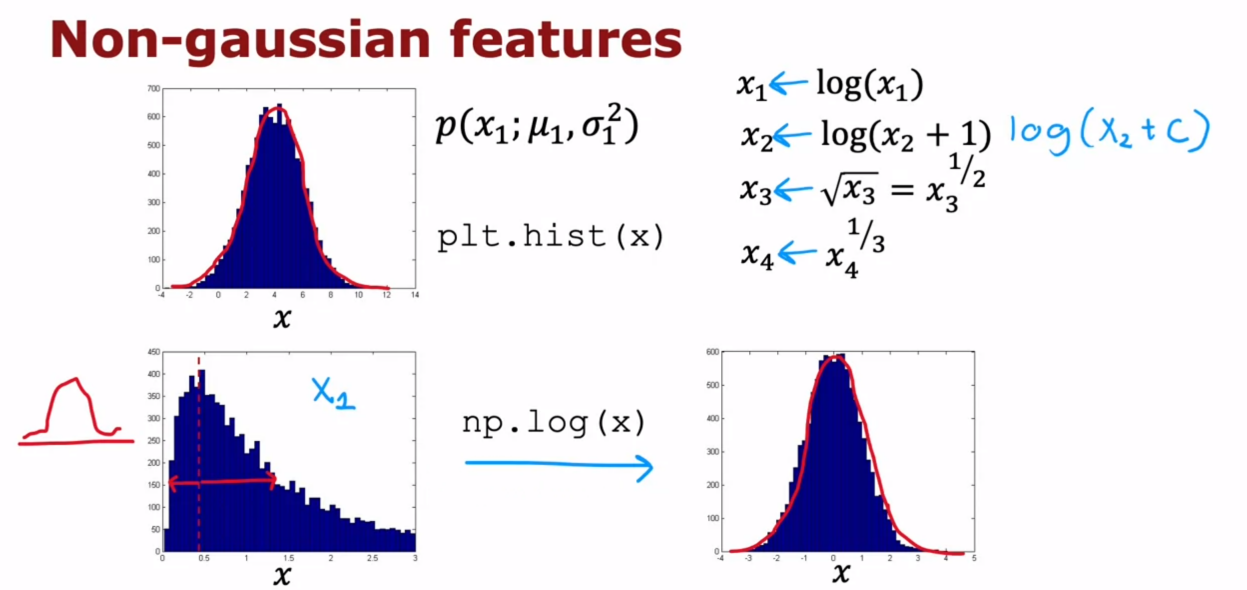
Examples:

- Fraud detection
- Manufacturing - Finding new previously unseen defects in manufacturing.(e.g. aircraft engines)
- Monitoring machines in a data center
- Email spam classification
- Weather prediction(eg. rainy/sunny)
- Manufacturing - Finding known, previously seen defects
- Diseases classification

13.5 Choosing What Features to Use

Non-Gaussian Features

If the features are not Gaussian distributed, we can try transforming the features to make them more Gaussian distributed. We want to make the features more Gaussian-like, so we can use the Gaussian model to detect anomalies.

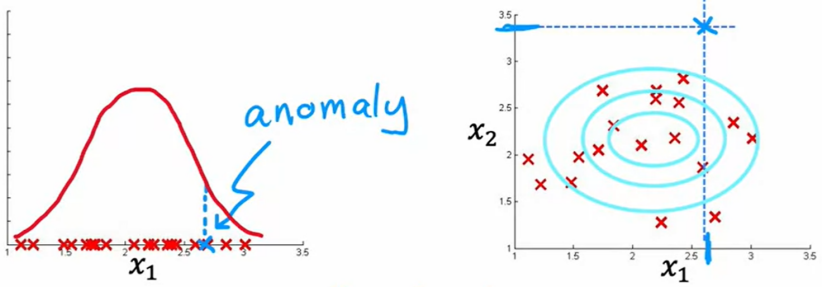


We want $p(x)$ to be large for normal examples and small for anomalies.

Error analysis for anomaly detection

Want $p(x) \geq \varepsilon$ large for normal examples x.
$p(x) < \varepsilon$ small for anomalous examples x.

Most common problem:
$p(x)$ is comparable for normal and anomalous examples.
($p(x)$ is large for both)



X_1 num transactions X_2 typing speed

We can combine the features to create new features.

Monitoring computers in a data center

Choose features that might take on
unusually large or small values
in the event of an anomaly.

x_1 = memory use of computer
x_2 = number of disk accesses/sec
x_3 = CPU load
x_4 = network traffic

high / low

not unusual

$$x_5 = \frac{\text{CPU load}}{\text{network traffic}} \qquad x_6 = \frac{(\text{CPU load})^2}{\text{network traffic}}$$

Deciding feature choice based on $p(x)$
Large for normal examples;
Becomes small for anomaly in the cross validation set

14 Collaborative Filtering

14.1 Making recommendations

Movie ratings example

	Alice	Bob	Carol	Dave
Love at last	5	5	0	0
Romance forever	5	?	?	0
Cute puppies of love	?	4	0	?
Nonstop car chases	0	0	5	4
Swords vs karate	0	0	5	?

Table 14.1: User rates movies using 0-5 stars

$$
\begin{aligned}
n_u &= \text{number of users} \\
n_m &= \text{number of movies} \\
r(i,j) &= 1 \text{ if user } j \text{ has rated movie } i \\
y(i,j) &= \text{rating given by user } j \text{ to movie } i
\end{aligned}
$$

In this case:

$n_u = 4, n_m = 5, r(1,1) = 1, y(3,1) = 0.$

14.2 Using per-item features

	Alice	Bob	Carol	Dave	x_1(romance)	x_2(action)
Love at last	5	5	0	0	0.9	0
Romance forever	5	?	?	0	1.0	0.01
Cute puppies of love	?	4	0	?	0.99	0
Nonstop car chases	0	0	5	4	0.1	1.0
Swords vs karate	0	0	5	?	0	0.9

Table 14.2: assuming we have features for each movie

For user j: Predict user j's rating for movie i as $w^{(j)} \cdot x^{(i)} + b^{(j)}$, where $w^{(j)}$ is the parameter vector for user j and $x^{(i)}$ is the feature vector for movie i. It's just like linear regression, but we have different parameters for each user.

Definition 14.2.1 ▸ Cost function

Notation:

$$\begin{aligned} r(i,j) &= 1 \quad \text{if user } j \text{ has rated movie } i \text{ (0 otherwise)} \\ y^{(i,j)} &= \text{rating by user } j \text{ on movie } i \text{ (if defined)} \\ x^{(i)} &= \text{feature vector for movie } i \\ w^{(j)}, b^{(j)} &= \text{parameter vector and bias for user } j \\ m^{(j)} &= \text{number of movies rated by user } j \end{aligned}$$

For user j, the cost function is:

$$J(w^{(j)}, b^{(j)}) = \frac{1}{2m^{(j)}} \sum_{i:r(i,j)=1} \left(w^{(j)} \cdot x^{(i)} + b^{(j)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2m^{(j)}} \sum_{k=1}^{n} (w_k^{(j)})^2 \tag{14.1}$$

The second term is the regularization term, n is the number of features.

Cost function for all users:

$$J\begin{pmatrix} w^{(1)} & \dots & w^{(n_u)} \\ b^{(1)} & \dots & b^{(n_u)} \end{pmatrix} = \frac{1}{2}\sum_{j=1}^{n_u}\sum_{i:r(i,j)=1}\left((w^{(j)})^T x^{(i)} + b^{(j)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}(w_k^{(j)})^2 \tag{14.2}$$

To learn $w^{(j)}, b^{(j)}$ for all users, minimize J, we can ignore the constant factor $m^{(j)}$.

14.3 Collaborative filtering algorithm

Cost function for x

Definition 14.3.1 ▸ Cost function

Cost function for single movie i:

$$J(x^{(i)}) = \frac{1}{2}\sum_{j:r(i,j)=1}\left(w^{(j)} \cdot x^{(i)} + b^{(j)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2}\sum_{k=1}^{n}(x_k^{(i)})^2 \tag{14.3}$$

Cost function for all movies:

$$J\left(x^{(1)}, \dots, x^{(n_m)}\right) = \frac{1}{2}\sum_{i=1}^{n_m}\sum_{j:r(i,j)=1}\left(w^{(j)} \cdot x^{(i)} + b^{(j)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}(x_k^{(i)})^2 \tag{14.4}$$

Collaborative filtering algorithm

Above cost function is for minimizing $w^{(j)}, b^{(j)}$ and $x^{(i)}$. Now, we can design a new algorithm to learn all these parameters simultaneously.

Theorem 14.3.2 ▸ Collaborative filtering

Cost function to learn $w^{(1)}, b^{(1)}, \cdots, w^{(n_u)}, b^{(n_u)}$:

$$\min_{w^{(1)},b^{(1)},\cdots,w^{(n_u)},b^{(n_u)}} \frac{1}{2}\sum_{j=1}^{n_u}\sum_{i:r(i,j)=1}\left(w^{(j)} \cdot x^{(i)} + b^{(j)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}(w_k^{(j)})^2$$

Cost function to learn $x^{(1)}, \cdots, x^{(n_m)}$:

$$\min_{x^{(1)},\cdots,x^{(n_m)}} \frac{1}{2}\sum_{i=1}^{n_m}\sum_{j:r(i,j)=1}\left(w^{(j)}\cdot x^{(i)}+b^{(j)}-y^{(i,j)}\right)^2+\frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}(x_k^{(i)})^2$$

To learn $w^{(j)}, b^{(j)}$ and $x^{(i)}$ simultaneously, we can minimize both cost functions:

$$\min_{\substack{w^{(1)},\cdots,w^{(n_u)}\\ b^{(1)},\cdots,b^{(n_u)}\\ x^{(1)},\cdots,x^{(n_m)}}} J(w,b,x)= \tag{14.5}$$

$$\frac{1}{2}\sum_{(i,j):r(i,j)=1}\left(w^{(j)}\cdot x^{(i)}+b^{(j)}-y^{(i,j)}\right)^2+\frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}(w_k^{(j)})^2+\frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}(x_k^{(i)})^2$$

Gradient descent

Having the cost function, we can use gradient descent to minimize it.

Theorem 14.3.3 ▶ Gradient descent

repeat{

$$w_i^{(j)} := w_i^{(j)} - \alpha\frac{\partial}{\partial w_i^{(j)}}J(w,b,x)$$

$$b^{(j)} := b^{(j)} - \alpha\frac{\partial}{\partial b^{(j)}}J(w,b,x)$$

$$x_k^{(i)} := x_k^{(i)} - \alpha\frac{\partial}{\partial x_k^{(i)}}J(w,b,x)$$

}

14.4 Binary labels

Previously: Predict $y^{(i,j)}$ as $w^{(j)}\cdot x^{(i)}+b^{(j)}$

For binary labels: Predict the probability that $y^{(i,j)}=1$ by $g(w^{(j)}\cdot x^{(i)}+b^{(j)})$, where $g(z)=\frac{1}{1+e^{-z}}$ is the sigmoid function.

Theorem 14.4.1 ▶ Cost function for binary labels

$$f_{(w,b,x)}(x) = g(w^{(j)} \cdot x^{(i)} + b^{(j)}) \tag{14.6}$$

$$L(f_{(w,b,x)}(x), y) = -y^{(i,j)} \log(f_{(w,b,x)}(x)) - (1 - y^{(i,j)}) \log(1 - f_{(w,b,x)}(x)) \tag{14.7}$$

$$J(w, b, x) = \sum_{(i,j):r(i,j)=1} L(f_{(w,b,x)}(x), y) \tag{14.8}$$

15 Implementation Details

15.1 Mean normalization

Consider this situation:

Users who have not rated any movies

Movie	Alice(1)	Bob (2)	Carol (3)	Dave (4)	Eve (5)
Love at last	5	5	0	0	?
Romance forever	5	?	?	0	?
Cute puppies of love	?	4	0	?	?
Nonstop car chases	0	0	5	4	?
Swords vs. karate	0	0	5	?	?

$$\min_{\substack{w^{(1)},\ldots w^{(n_u)}\\ b^{(1)},\ldots b^{(n_u)}\\ x^{(1)},\ldots x^{(n_m)}}} \frac{1}{2}\sum_{(i,j):r(i,j)=1}\left(w^{(j)}\cdot x^{(i)}+b^{(j)}-y^{(i,j)}\right)^2+\frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}\left(w_k^{(j)}\right)^2+\frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}\left(x_k^{(i)}\right)^2$$

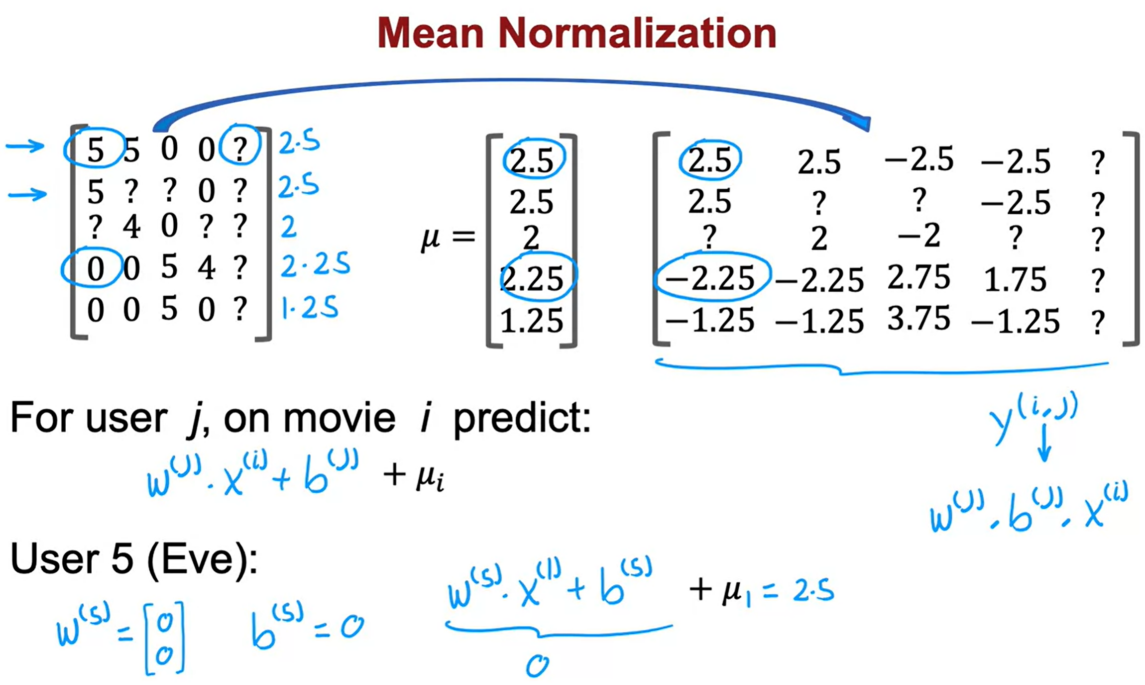
$w^{(5)}=\begin{bmatrix}0\\0\end{bmatrix}$ $b^{(5)}=0$ $w^{(5)}\cdot x^{(i)}+b^{(5)}$

Eve didn't rate any movies, so her predicted ratings are all 0. Because of the regularization term, her parameters are all 0. So, her predicted ratings for all movies are 0.

To avoid this, we can use mean normalization. Subtract the mean rating of a movie from the user's rating. And add the mean rating to the predicted rating.

- Represent the data as a matrix Y where $Y_{i,j}$ is the rating of movie i by user j.
- Compute the mean of the ratings of movie i (excluding missing values), μ_i. This is same to say compute the mean of each row of the matrix Y.
- For each user j who rated movie i, set $Y_{i,j} = Y_{i,j} - \mu_i$.
- Learn parameters on this new matrix.

- To predict the rating of movie i by user j, add μ_i to the predicted value.$(w^{(j)} \cdot x^{(i)} + b^{(j)} + \mu_i)$



In the above example, for user who didn't rate any movies, the predicted ratings are not 0 but the mean rating of the movie.

The mean normalization can be also applied to the column of the matrix Y, while in this case, it is more appropriate to use the mean of the user's ratings. In practical applications, we choose which mean to use based on the context.

15.2 Implementation in TensorFlow

TensorFlow has a powerful tool to automatically compute the derivatives of a function with respect to its parameters. This is called **auto diff (automatic differentiation)**.

Code Snippet 15.2.1 ▸ auto diff

```
w = tf.Variable(3.0)
x = 1.0
y = 1.0
alpha = 0.01

iterations = 30
for iter in range(iterations):
    with tf.GradientTape() as tape:
        fwb = w * x
        cost = (fwb - y) ** 2
    grad = tape.gradient(cost, w)
    w.assign(w - alpha * grad)
```

Recall that there is an optimizer called "Adam" in TensorFlow. It can change the learning rate during training.

Code Snippet 15.2.2 ▸ Adam optimizer

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)

iterations = 200
for iter in range(iterations):
    with tf.GradientTape() as tape:
        cost = cofiCostFunc(X, w, b, Ynorm, R,
            num_users, num_movies, num_features, lambda_)
    grads = tape.gradient(cost, [X, w, b])
    optimizer.apply_gradients(zip(grads, [X, w, b]))
```

15.3 Related items

The feature $x^{(i)}$ of item i can show what the item is about. To find other items related to item i, we can find the items with similar features.

i.e. Find item k such that $||x^{(i)} - x^{(k)}||$ is small. (Euclidean distance or norm) $||x^{(k)} - x^{(i)}||^2 = \sum_{l=1}^{n} \left(x_l^{(k)} - x_l^{(i)}\right)^2$.

Limitations of collaborative filtering

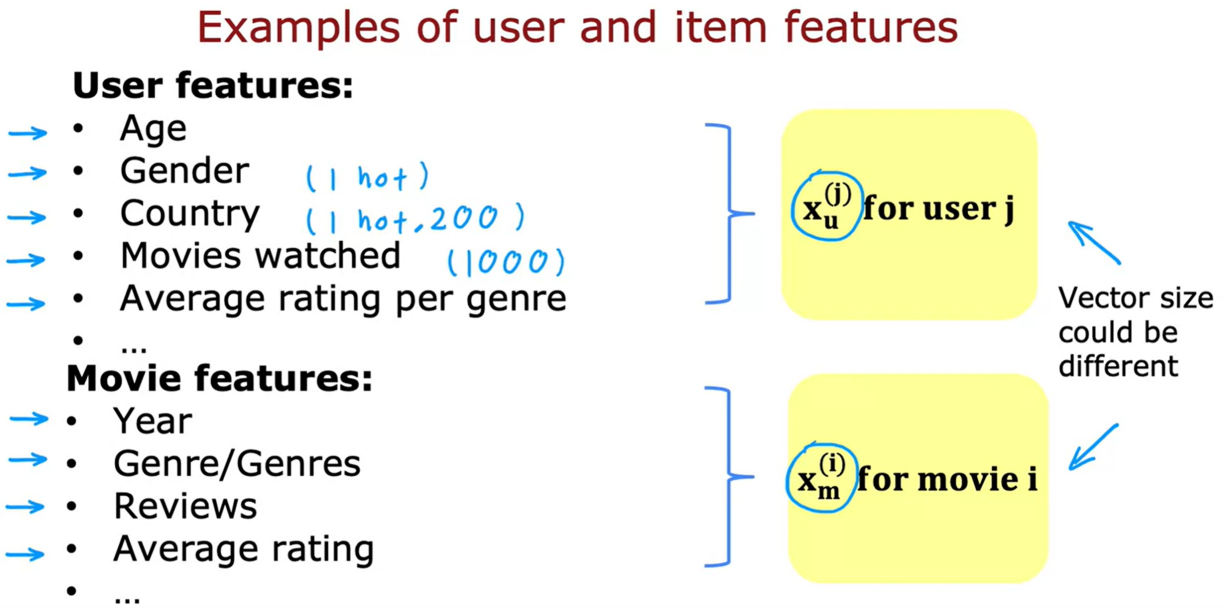
- Cold start problem:
  - It's hard to recommend new items because there are no ratings for them.
  - It's hard to recommend items to new users because there are few ratings from them.
- Use side information about items or users:
  - item: movie genres, movie stars, movie directors, etc.
  - user: Demographic information (age, gender, etc.), social network information, etc.

16 Content-based filtering

16.1 Prediction

Conten-based filtering is based on features of user and item features. The same definition as before: $r(i, j) = 1$ if user j has rated movie i (0 otherwise). $y(i, j)$ is the rating of item i by user j.

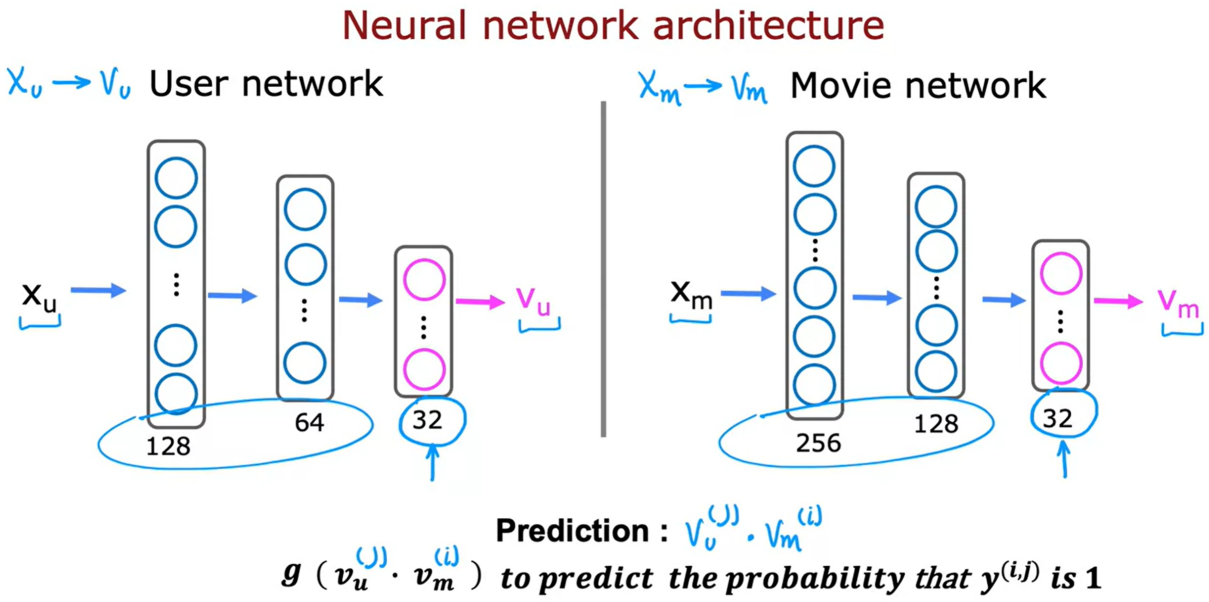
However, there is no need to learn parameters $w^{(j)}$ and $b^{(j)}$ for each user j. Instead, we use a user features vector $\mathbf{x}_u^{(j)}$ to represent user j. Similarly, we use an item features vector $\mathbf{x}_m^{(i)}$ to represent movie i. Note that the dimensions of the user and item features vectors may be different.



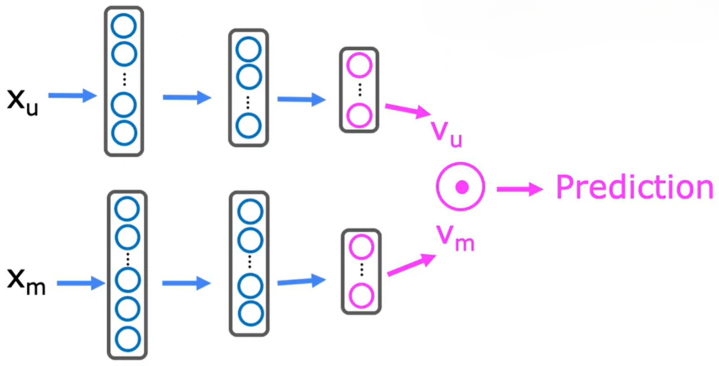
Predict rating of movie i by user j: $\mathbf{v}_u^{(j)} \cdot \mathbf{v}_m^{(i)}$. The $\mathbf{v}_u^{(j)}$ is computed from the user features vector $\mathbf{x}_u^{(j)}$ and the $\mathbf{v}_m^{(i)}$ is computed from the item features vector $\mathbf{x}_m^{(i)}$. Because we need to take dot product of two vectors, the dimensions of the two vectors must be the same.

Nerual network architecture

To compute the $\mathbf{v}_u^{(j)}$ and $\mathbf{v}_m^{(i)}$, we can use a neural network architecture.



By taking the dot product of the output of the two neural networks, we can predict the rating of movie i by user j. Because we want to keep the dimensions of the two vectors the same, we need to set the number of units in the output layer of the two neural networks to be the same. This is a good example of combining two separate neural networks to make a prediction. (That's one of the advantages of neural networks compared to random forests model.)



Similarly, set the cost function J to be:

$$J = \sum_{(i,j):r(i,j)=1} \left(\mathbf{v}_u^{(j)} \cdot \mathbf{v}_m^{(i)} - y(i,j)\right)^2 + \text{NN regularization terms}$$

To find movie k that are similar to movie i, we can compute the distance between the item features vectors $\mathbf{x}_m^{(i)}$ and $\mathbf{x}_m^{(k)}$: $||\mathbf{v}_m^{(k)} - \mathbf{v}_m^{(i)}||$, the smaller the distance, the more similar the two movies are.

16.2 Recommendations

1. **Retrieval**
   (a) Generate large lists of plausible item candidates:
      i. For each of the last 10 movies watched by the user, find 10 similar movies.
      ii. For most viewed 3 genres, find 10 top movies.
      iii. Find top 20 popular movies in the country.
      iv. ⋯
   (b) Combine retrieved items into list, removing duplicates and items already watched.
2. **Ranking**
   (a) Take retrieved list and rank items by predicted ratings using the model.
   (b) Display top N items to user.

Retrieving more items results in better performance, but slower recommendations. To analyse/optimize the trade-off, carry out offline experiments to see if retrieving additional items results in more relevant recommendations (i.e., $p\left(y^{(i,j)}\right) = 1$ of items displayed to user are higher).

Ethical use of recommender systems

Other problematic cases:

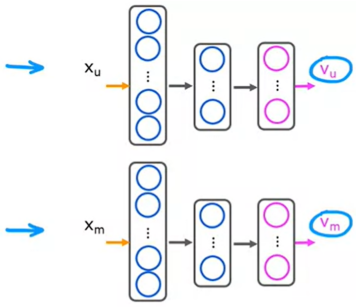
- Maximizing user engagement (e.g. watch time) has led to large social media/video sharing sites to amplify conspiracy theories and hate/toxicity

  → Amelioration : Filter out problematic content such as hate speech, fraud, scams and violent content

- Can a ranking system maximize your profit rather than users' welfare be presented in a transparent way?

  Amelioration : Be transparent with users

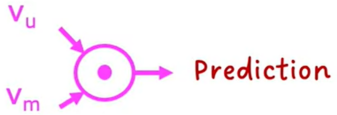
16.3 Implementation in TensorFlow




```
user_NN = tf.keras.models.Sequential([
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(32)
])
```

```
item_NN = tf.keras.models.Sequential([
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(32 )
])
```

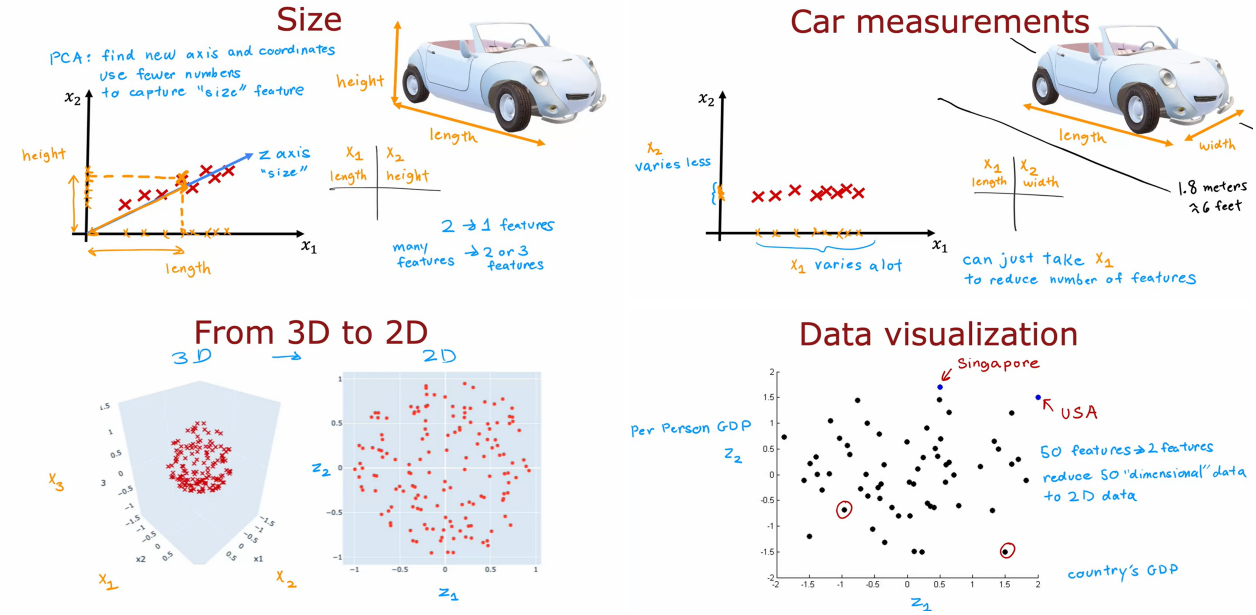
```
# create the user input and point to the base network
input_user = tf.keras.layers.Input(shape=(num_user_features))
vu = user_NN(input_user)
vu = tf.linalg.l2_normalize(vu, axis=1)

# create the item input and point to the base network
input_item = tf.keras.layers.Input(shape=(num_item_features))
vm = item_NN(input_item)
vm = tf.linalg.l2_normalize(vm, axis=1)

# measure the similarity of the two vector outputs
output = tf.keras.layers.Dot(axes=1)([vu, vm])

# specify the inputs and output of the model
model = Model([input_user, input_item], output)

# Specify the cost function
cost_fn = tf.keras.losses.MeanSquaredError()
```



17 Principal Component Analysis

17.1 Reducing the number of the features

What PCA does is to find a lower-dimensional surface onto which to project the data, so as to minimize the projection error. By using PCA, we can reduce the dimension of the data, and make it easier to visualize and understand.



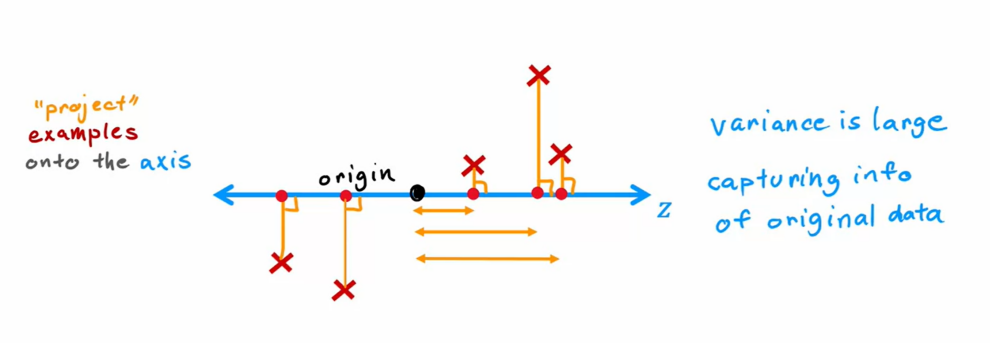
17.2 PCA Algorithm

Data preprocessing

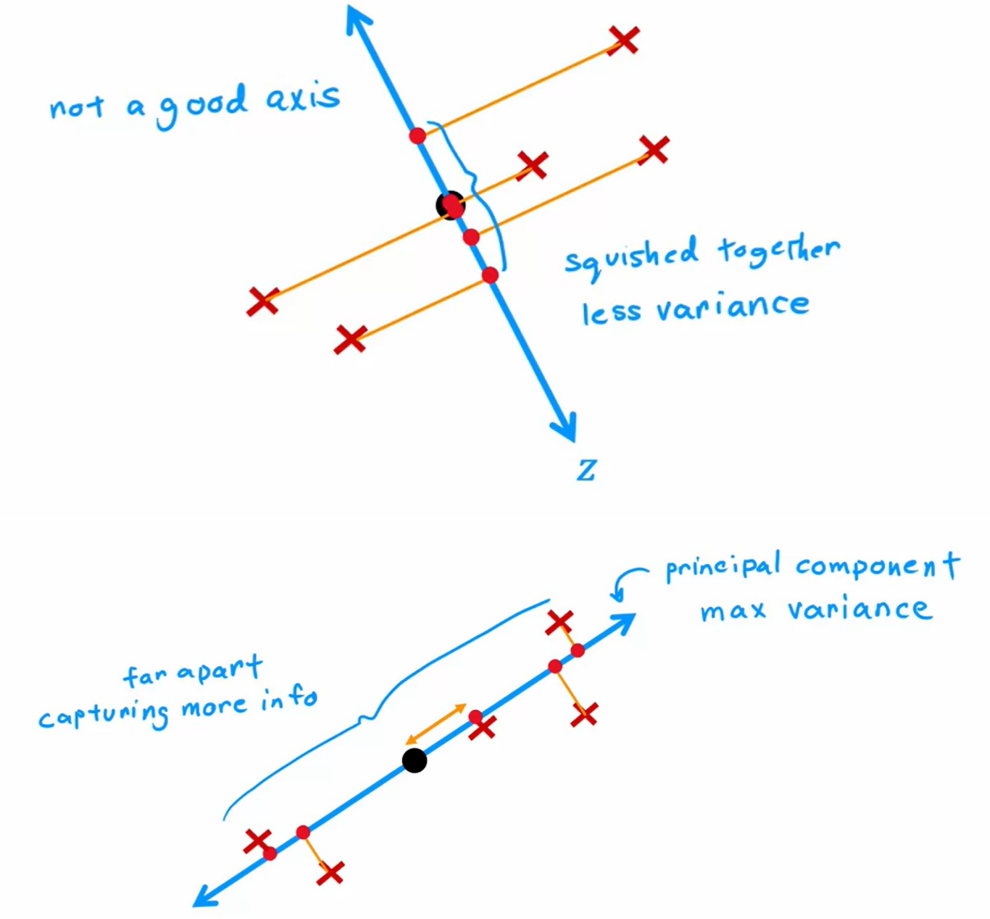
Before applying PCA, it is important to first normalize to have zero mean. Apply feature scaling and mean normalization to the data so that each feature has a equal influence on the computation of the principal components. (for example, min-max normalization or z-score normalization)

Projection

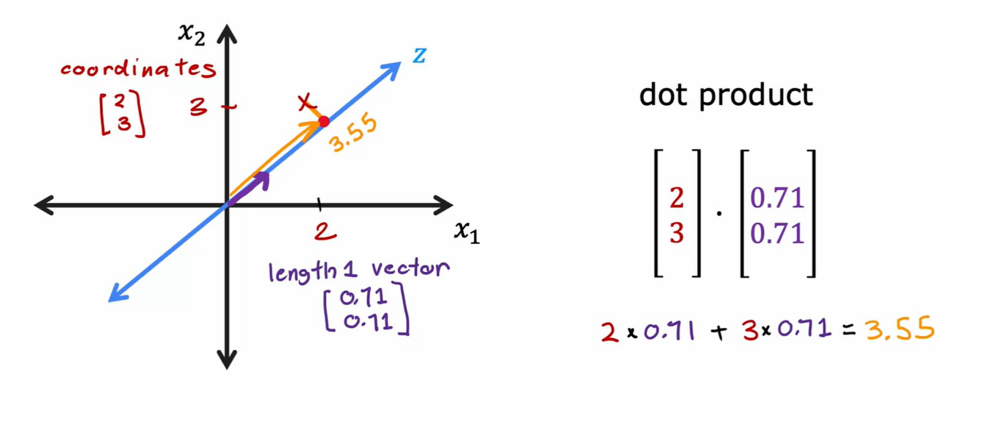
"Project" examples onto the new axis. The new axis are the principal components. It should have variance as large as possible so that can capture info of the orignal data.



Looking at the 2D plot, we can see that the data is spread out along the direction of the principal component.



To calculate the coordinates of the data points in the new basis, we can use the dot product: assuming $\mathbf{x}$ is the orignal data, and $\mathbf{u}$ is the principal component, and $\mathbf{u}$ is the unit vector which has a length of 1. Then the length of projection of $\mathbf{x}$ onto $\mathbf{u}$ is $\mathbf{x} \cdot \mathbf{u}$. And we can multiply $\mathbf{u}$ by $\mathbf{x} \cdot \mathbf{u}$ to get the projection vector. This step is called reconstruction.



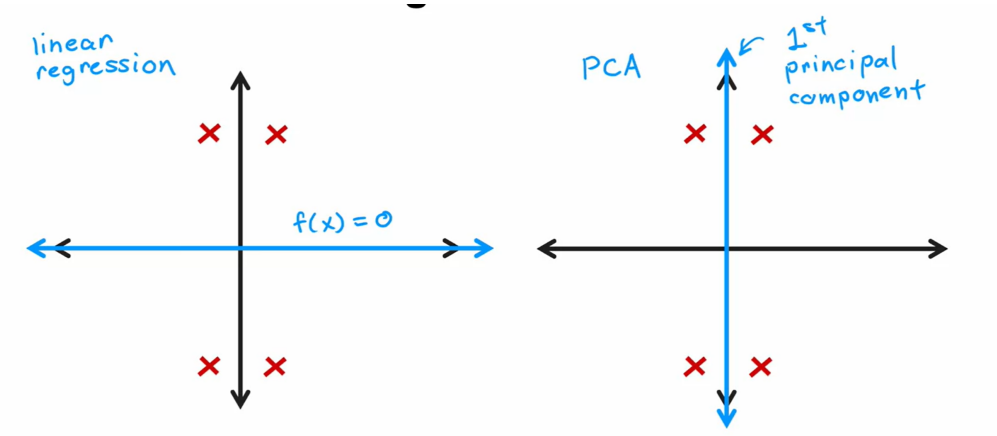
If there are pricipal components more than 1, we can project the data onto the first k principal components. We call them the first k principal components. And each of them is orthogonal to the others.

Difference between PCA and Linear Regression

- In linear regression, we are trying to minimize the error between the prediction and the actual value.
- In PCA, we are trying to minimize the projection error.

And the difference is more obvious in higher dimensions.

Below is one example of the difference between PCA and linear regression:



17.3 PCA in scikit learn

> the “fit” method in sklearn already includes the step of mean normalization.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

X = np.array([[1, 1], [2, 1], [3, 2], [-1, -1], [-2, -1], [-3, -2]])
plt.scatter(X[:, 0], X[:, 1])
plt.show()

pca = PCA(n_components=1)
pca.fit(X)
pca.explained_variance_ratio_

X_pca = pca.transform(X)
X_pca

inverse = pca.inverse_transform(X_pca)
plt.scatter(inverse[:, 0], inverse[:, 1])
plt.show()
```

Application of PCA

- Data compression
- Visualization
- Speeding up learning algorithms

18 Reinforcement Learning Introduction

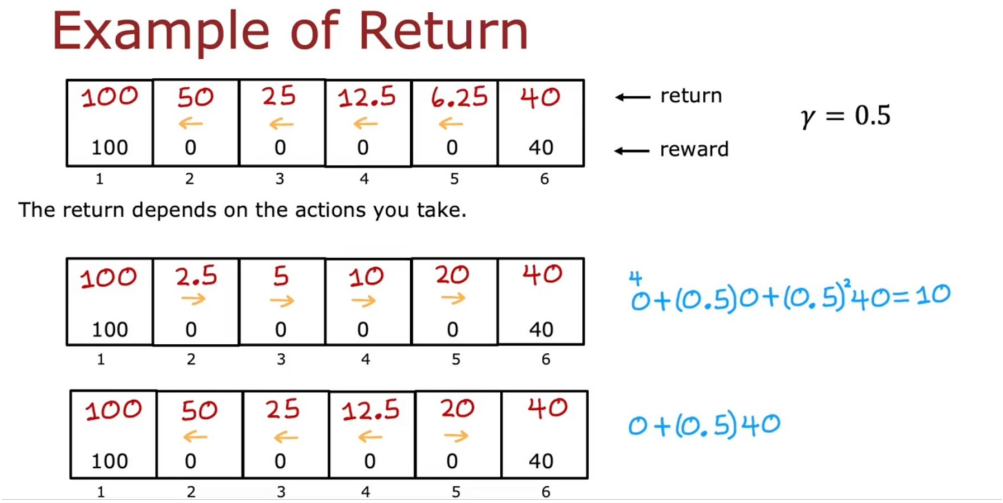
18.1 What is reinforcement learning?

Reinforcement learning is a type of machine learning that allows us to create agents that learn to take actions in an environment in order to maximize some notion of cumulative reward. The agent learns to achieve a goal in an uncertain, potentially complex environment. In reinforcement learning, an agent interacts with an environment by taking actions and receiving feedback, in the form of rewards or penalties. The agent then uses this feedback to learn how to interact with the environment in the future.

Below are some imortant concepts in reinforcement learning:

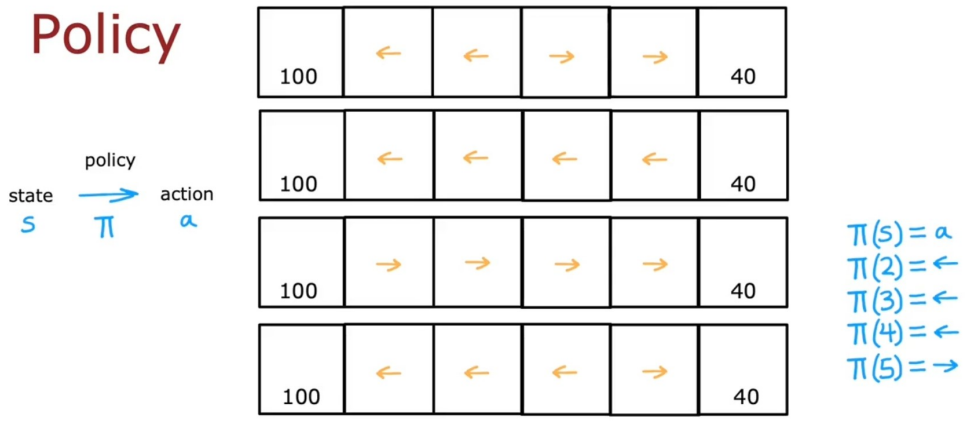
Return

The return is the sum of rewards that the agent receives over time. The return depends on the sequence of actions taken by the agent.



Policy

The policy is a function $\pi(s) = a$ mapping from states to actions, which tells you what action a to take in a given state s. The policy can be deterministic or stochastic.



Discount factor

The discount factor γ is a value between 0 and 1 that determines how much the agent values future rewards. If γ is close to 0, the agent will focus on immediate rewards. If γ is close to 1, the agent will be more concerned with long-term rewards.

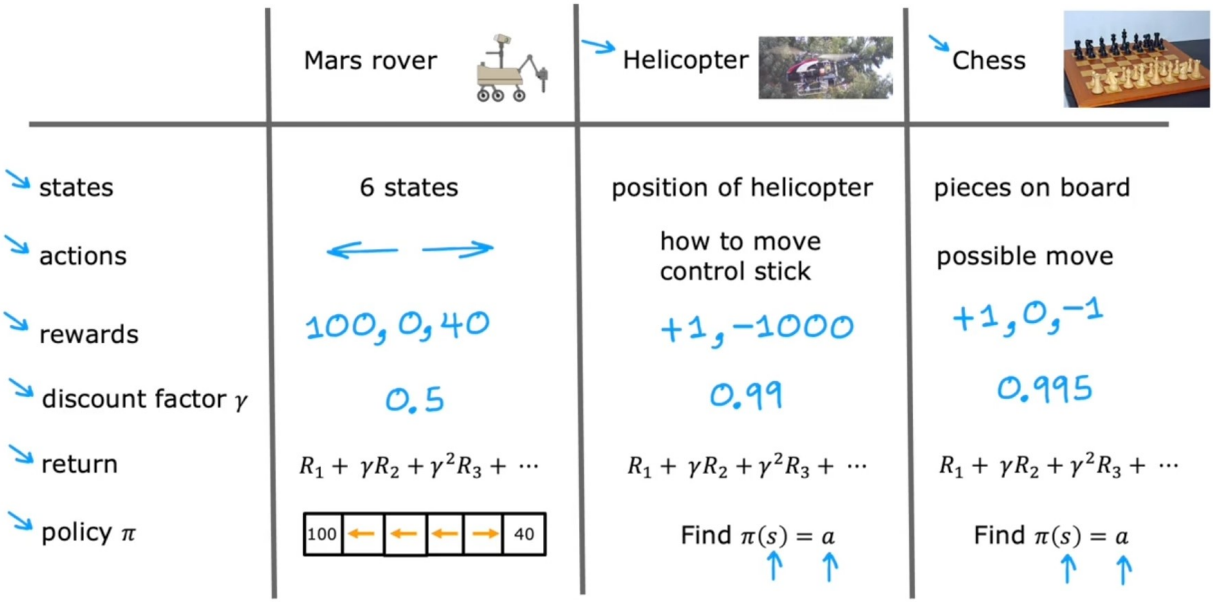
Rewards

The rewards are the feedback that the agent receives from the environment. Rewards often means the immediate response that the agent get being in a state.

The goal of reinforcement learning

The goal of reinforcement learning is to find the policy π that maximizes the expected return.

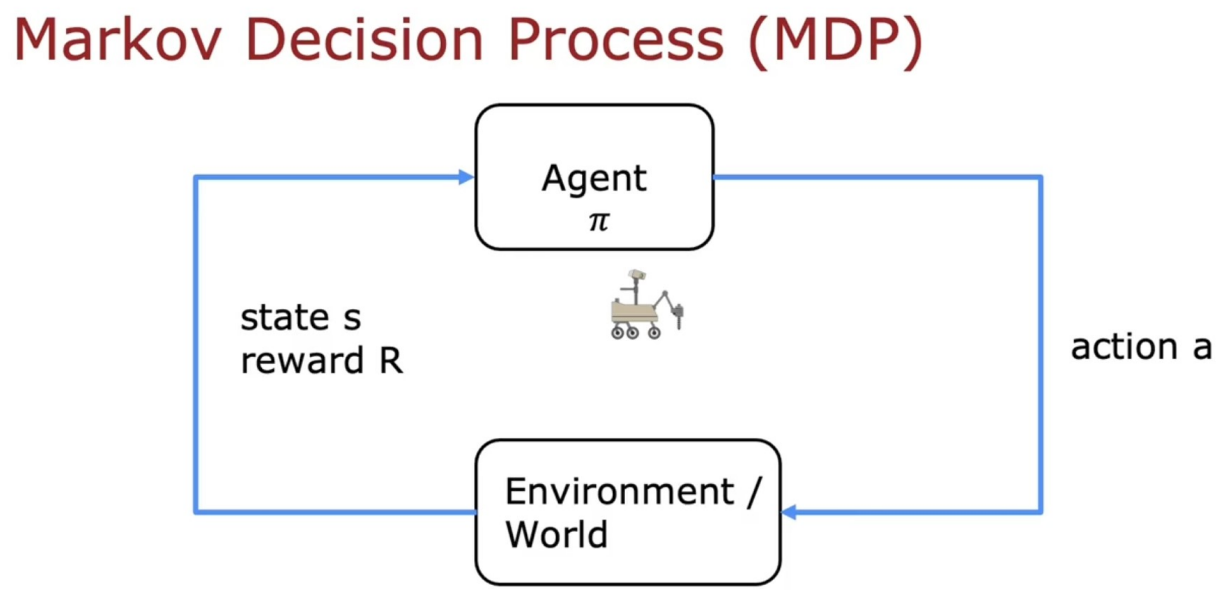
Some examples of reinforcement learning



18.2 Markov Decision Process

A Markov Decision Process (MDP) is a mathematical framework for modeling decision-making in situations where outcomes are partly random and partly under the control of a decision maker. An MDP consists of the following components:

- An environment with a set of states S.
- An agent with a policy π and a set of actions A.
- A transition function $P(s, a, s')$ that gives the probability of transitioning from state s to state s' by taking action a.
- A reward function $R(s, a, s')$ that gives the reward received for transitioning from state s to state s' by taking action a.
- A discount factor γ.

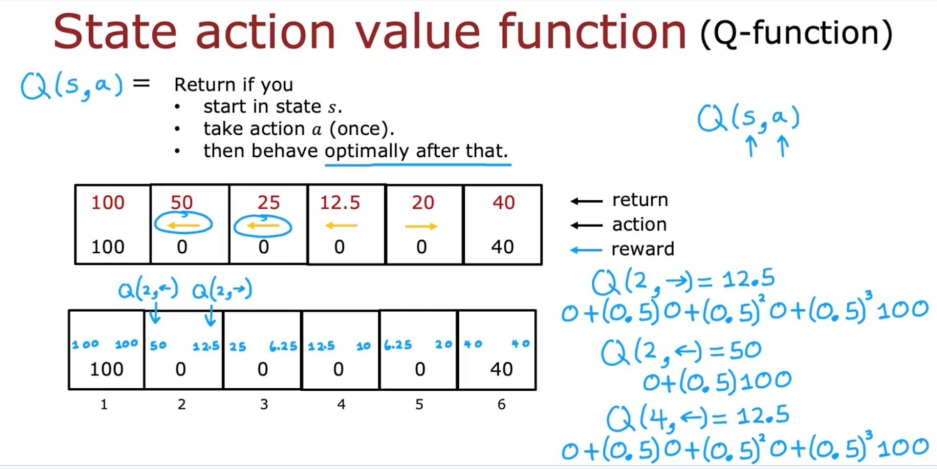


18.3 State action value function

Definition 18.3.1 ▸ Q-function

The state-action value function $Q(s, a)$ is the expected return of taking action a in state s. It is also called the Q-function.

$Q(s, a)$ == Return of state s iff start in state s, take action a (once), then behave optimally after that.



Picking actions

The best possible return from state s is $\max_a Q(s,a)$, The best possible action in state s is $\operatorname{argmax}_a Q(s,a)$. Optimal Q function is noted as $Q^*(s,a)$.

18.4 Bellman Equation

Theorem 18.4.1 ▸ Bellman Equation

s current state
a current action
s' state after taking action a in state s
a' action in state s'
$R(s)$ reward of state s

$$Q(s,a) = R(s) + \gamma \max_{a'} Q(s',a') \tag{18.1}$$

Stochastic environment

Stochastic environment means that the transition function $P(s,a,s')$ is not deterministic, which is to say that the mars rover may sometimes slip on a rock and not move as expected.

The definition of "Return" has a more general form:

$$\text{Expected Return} = E\left[R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \cdots\right] \tag{18.2}$$

The E means the expectation of the sum of rewards. The rewards are random variables, so the return is also a random variable.

In the Bellman Equation, s' is also a random variable, here is the general version of the Bellman Equation:

$$Q(s,a) = R(s) + \gamma E\left[\max_{a'} Q(s',a')\right] \tag{18.3}$$

19 Continuous State Space

19.1 Continuous state examples

Continuous state space is a state space where the state can take on any real value within a range. The numbers of states in a continuous state space are infinite, which makes it impossible to represent the state space as a table. To model it, we can represent each state as a vector of real numbers, and use a function to approximate the Q-function.

In the truck example, the state space is continuous because the truck can be at any position on the road.

$$s = \begin{bmatrix} \text{Position} \\ \text{Speed} \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \\ \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}$$

In this example, x, y is the position of the truck, θ is the angle of the truck, $\dot{x}$, $\dot{y}$, $\dot{\theta}$ is the corresponding speed of the truck.

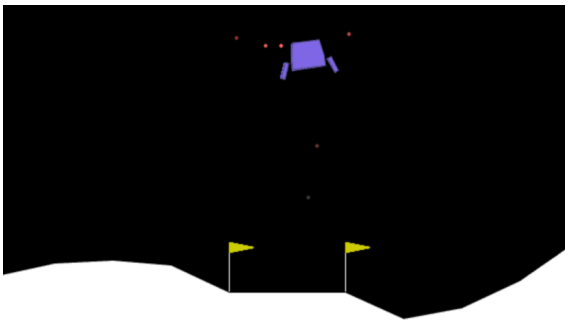
In the aircraft example, the state space:

$$s = \begin{bmatrix} \text{Position} \\ \text{Heading} \\ \text{Speed} \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \phi \\ \theta \\ \omega \\ \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\omega} \end{bmatrix}$$

In this example, x, y is the position of the aircraft horizontally, z is the height, ϕ is the roll angle, θ is the pitch angle, ω is the yaw angle, $\dot{x}$, $\dot{y}$, $\dot{z}$ is the corresponding speed of the aircraft, $\dot{\phi}$, $\dot{\theta}$, $\dot{\omega}$ is the corresponding angular velocity of the aircraft.

Lunar lander example

The lunar lander example is a classic reinforcement learning problem.



The available actions are:

- Do nothing

- left thruster
- main thruster
- right thruster

The state space is continuous, the state is represented as a vector:

$$s = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \theta \\ \dot{\theta} \\ \text{l} \\ \text{r} \end{bmatrix}$$

x, y is the position of the lunar lander, $\dot{x}$, $\dot{y}$ is the speed of the lunar lander, θ is the shifting angle of the lunar lander, $\dot{\theta}$ is the angular velocity of the lunar lander, l and r means the left and right leg of the lunar lander, 1 if the leg is touching the ground, 0 otherwise.

Reward function

The reward function is defined as:

- Getting to the landing pad: +100 points
- Additional reward for moving toward/away from the landing pad.
- Crash: -100 points
- Soft landing: +100 points
- Leg contact with the ground: +10 points
- Using the main engine: -0.3 points
- Using the side engine: -0.03 points

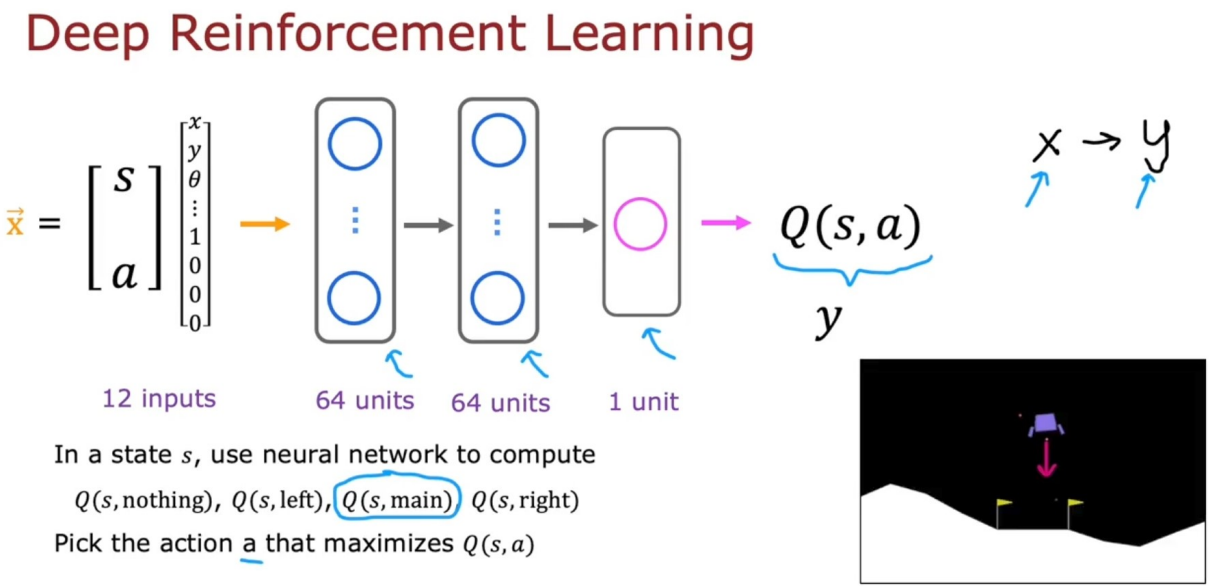
Goal

Learn a policy π that given

$$s = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \theta \\ \dot{\theta} \\ \text{l} \\ \text{r} \end{bmatrix} \qquad \gamma = 0.985$$

picks action $a = \pi(s)$ so as to maximize the return.

19.2 Deep reinforcement learning



The input is the combination of the state and the action. State is the same as the state vector in the continuous state space, and the action is the one-hot encoding of the action. The output is the Q-value of the state-action pair.

Like supervised learning, we can use a neural network to approximate the Q-function. $\mathbf{x}$ is the input, y is $Q(s,a)$. Neural network can learn the mapping $\mathbf{x} \to y$.

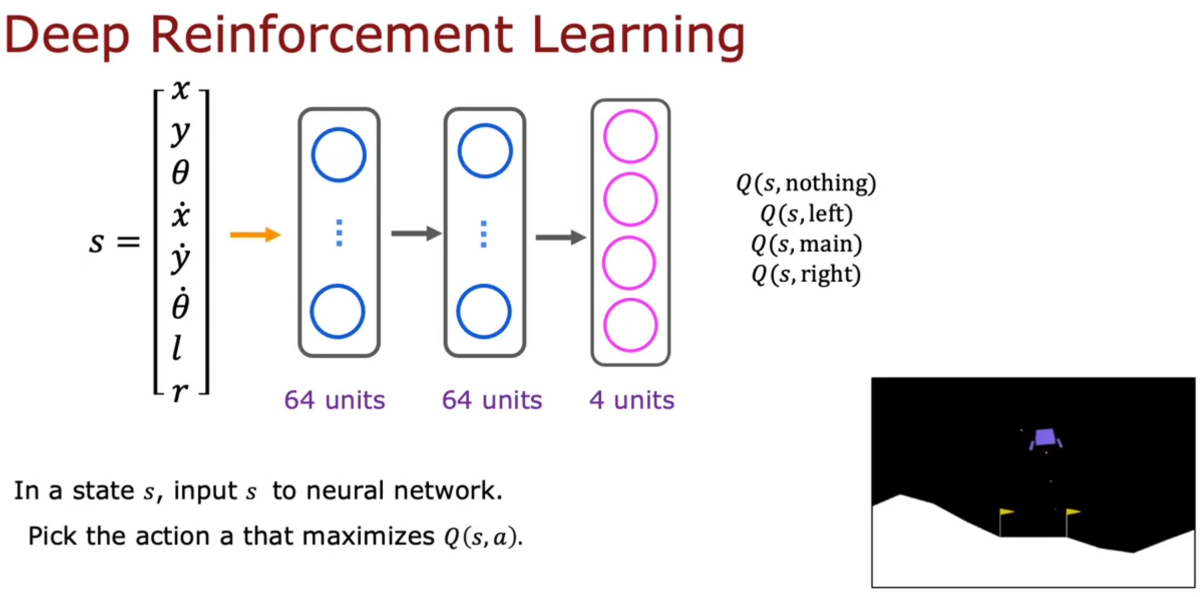
Learning algorithm

Theorem 19.2.1 ▸ Deep Q-learning (DQN)

- Initialize nerual network randomly as guess of $Q(s,a)$
- Repeat:
  - Take actions in the lunar lander, get tuple $(s, a, R(s), s')$
  - Store 10,000 most recent $(s, a, R(s), s')$ tuples (**Replay Buffer**)
  - Train nerual network:
    - Create training set of 10,000 examples using $x = (s,a)$ and $y = R(s) + \gamma \max_{a'} Q(s', a')$
    - Train Q_{new} such that $Q_{\text{new}}(s,a) \approx y$
  - Update $Q(s,a) \leftarrow Q_{\text{new}}(s,a)$

19.3 Algorithm refinement

Improved neural network architecture



The input is just the state vector, and the output is multiple Q-values for each action. This

architecture is more efficient than the previous one, because the previous one has to calculate the Q-value for each action separately and then combine them. Meanwhile, this architecture can better combine with the Bellman equation formula.

ε-greedy policy

How to choose actions while still learning?

Theorem 19.3.1 ▶ ε-greedy

In some state, apply ε-greedy policy ($\varepsilon = 0.05$)

Exploraion With probability $\varepsilon(0.05)$, pick an action a randomly.

Exploitation With probability $1 - \varepsilon(0.95)$, pick the best action $a = \underset{a}{\operatorname{argmax}}\, Q(s, a)$.

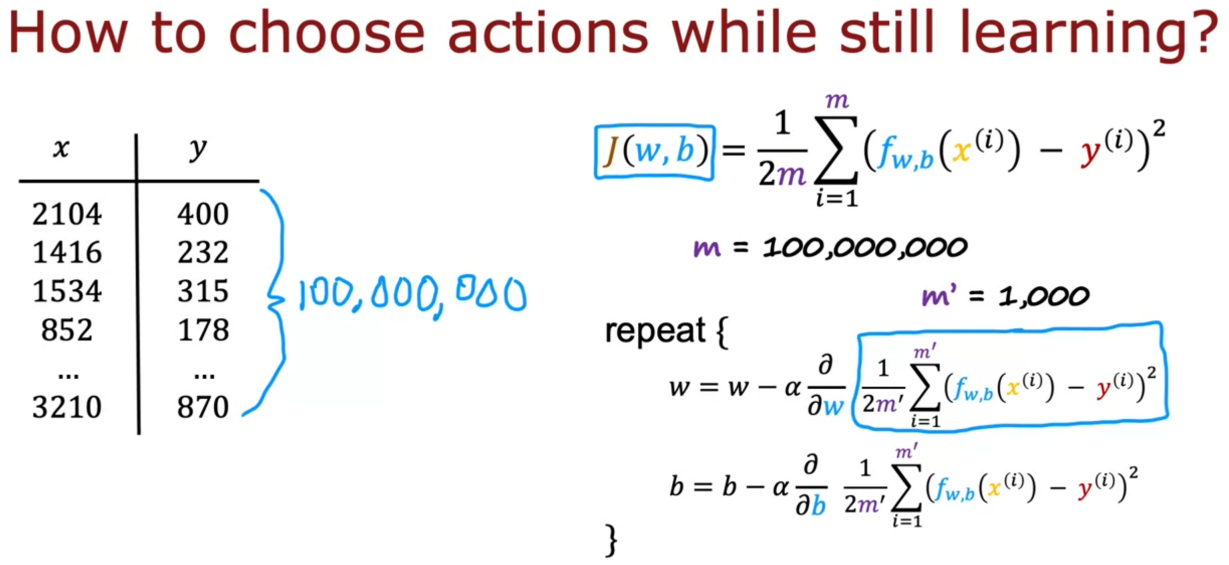
Although the policy is called ε-greedy, the "greedy" part is $1 - \varepsilon$ (0.95).

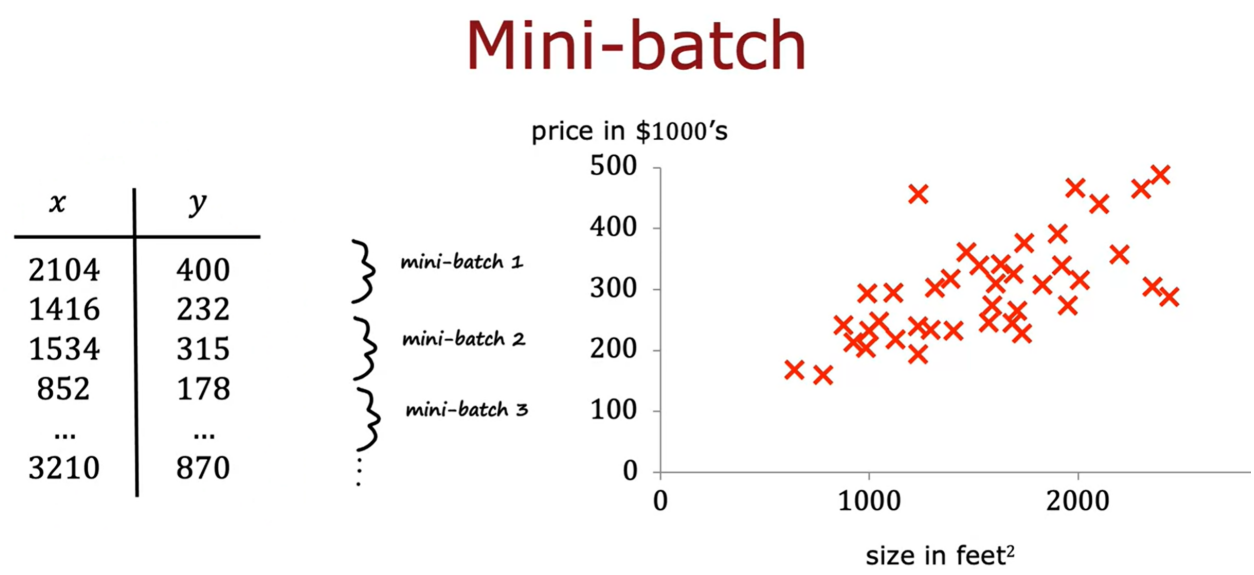
To get better results, anneal ε over time. Start with $\varepsilon = 1$, decrease ε over time.

Mini-batch

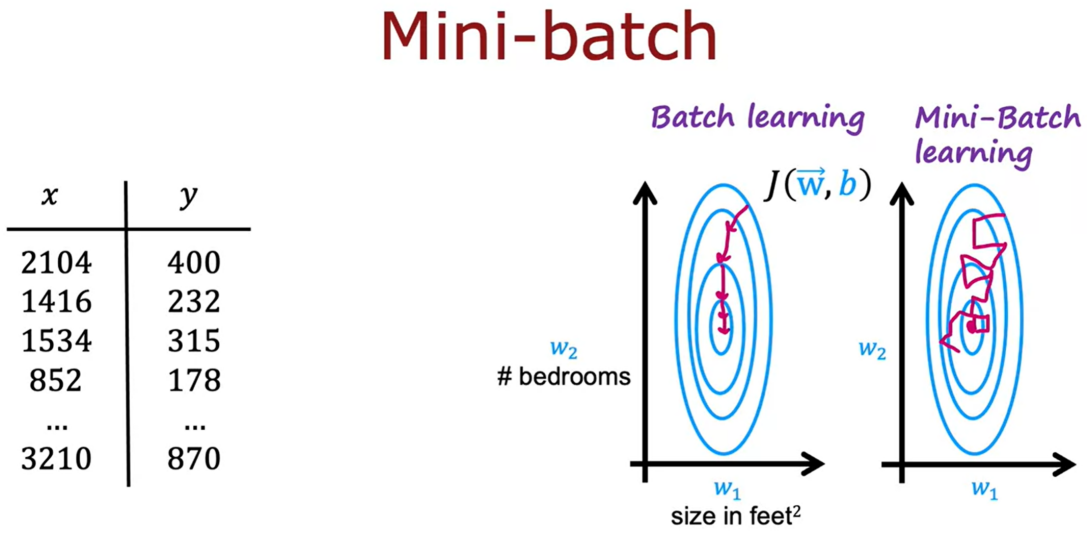
Mini-batch training is used both in supervised learning and reinforcement learning.

When the training set is large, the training process is computatuionally expensive and slow. Hence, we can use mini-batch training to speed up the training process. Mini-batch is a subset of the training set. Instead of running gradient descent on the entire training set, we run it on a mini-batch.





The learning curve of mini-batch training is not as smooth as the full batch training, the cost may increase at some ponit, but it will finally converges to the same result as the full batch training.



Soft update

Theorem 19.3.2 ▶ Soft update

- Update the target network slowly
- Instead of updating the target network with the Q_{new} directly, we update the target network with a fraction of the Q_{new}.
- $W := 0.99W + 0.01W_{\text{new}}$
- $B := 0.99B + 0.01B_{\text{new}}$

This will make the training process more stable.

Index

Definitions

Examples

Theorems

Code Snippets